NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

SIMULATION FOR SMARTNET SCHEDULING OF ASYNCHRONOUS TRANSFER MODE VIRTUAL CHANNELS

by

Michael J. Lemanski
and
Jesse C. Benton

June 1997

Thesis Advisor: Debra Hensgen

Approved for public release; Distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE June 1997	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE **SIMULATION FOR SMARTNET SCHEDULING OF ASYNCHRONOUS TRANSFER MODE VIRTUAL CHANNELS**

5. FUNDING NUMBERS

6. AUTHOR(S)
Lemanski, Michael J.
Benton, Jesse C.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13.ABSTRACT ***(maximum 200 words)*** Critical to the success of all future battlefield commanders is the rapid retrieval of relevant, time sensitive information. Some of this information will be available locally while the remainder is stored in the United States. DARPA's Battlefield Awareness and Data Dissemination (BADD) program attempts to deliver heterogeneous data to the battlefield using Asynchronous Transfer Mode (ATM) protocol. ATM was originally designed to implement dynamic virtual channels over duplex, high-speed, high capacity fiber optic cabling. The problem addressed was to determine which algorithm best schedules calls on BADD's ATM network that uses static virtual channels over simplex, error prone, long delay, satellite links. Because the BADD project uses ATM in such an unusual way, and because of the need to determine a schedule for transmissions over the heterogeneous static channels, we modeled BADD using the state-of-the-art network simulation tool, Optimized Network Engineering Tools (OPNET). We determined several modifications that must be made to existing network simulators to allow them to model next-generation networks. Our simulation shows that a greedy algorithm yields a 53% decrease in the overall completion time and a 46% increase in average bit throughput over FIFO scheduling.

14. SUBJECT TERMS
The Battlefield Awareness and Data Dissemination (BADD) , Asynchronous Transfer Mode (ATM), scheduling

15. NUMBER OF PAGES 290

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

Approved for public release; Distribution is unlimited.

SIMULATION FOR SMARTNET SCHEDULING OF ASYNCHRONOUS TRANSFER MODE VIRTUAL CHANNELS

Michael J. Lemanski
Captain, United States Army
B.S., United States Military Academy, 1986
and
Jesse C. Benton
Captain, United States Marine Corps
B.S., University of New Mexico, 1988

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE
from the
NAVAL POSTGRADUATE SCHOOL
June 1997

Authors:

Michael J. Lemanski

Jesse C. Benton

Approved by:

Debra Hensgen, Thesis Advisor

Geoffrey Xie, Second Reader

Ted Lewis, Chairman, Department of Computer Science

ABSTRACT

Critical to the success of all future battlefield commanders is the rapid retrieval of relevant, time sensitive information. Some of this information will be available locally while the remainder is stored in the United States. DARPA's Battlefield Awareness and Data Dissemination (BADD) program attempts to deliver heterogeneous data to the battlefield using Asynchronous Transfer Mode (ATM) protocol. ATM was originally designed to implement dynamic virtual channels over duplex, high-speed, high capacity fiber optic cabling. The problem addressed was to determine which algorithm best schedules calls on BADD's ATM network that uses static virtual channels over simplex, error prone, long delay, satellite links. Because the BADD project uses ATM in such an unusual way, and because of the need to determine a schedule for transmissions over the heterogeneous static channels, we modeled BADD using the state-of-the-art network simulation tool, Optimized Network Engineering Tools (OPNET). We determined several modifications that must be made to existing network simulators to allow them to model next-generation networks. Our simulation shows that a greedy algorithm yields a 53% decrease in the overall completion time and a 46% increase in average bit throughput over FIFO scheduling.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

I. INTRODUCTION

A. BACKGROUND

From several years before Operation Urgent Fury, the invasion in Grenada in 1983, until after Operation Just Cause, the invasion in Panama in 1989, the Department of Defense (DoD) fielded numerous new stand-alone automated systems. These systems provided solutions to narrowly focused problems and were not interoperable with other DoD systems. The story of the enterprising young Army officer in Grenada, calling his home base in North Carolina to coordinate naval gunfire support, because his equipment could not directly communicate with naval equipment, illustrates the problems encountered by DoD personnel in using such systems. Today these systems are referred to as **"vertical"** because they do not facilitate **"horizontal"** communication. (They are also sometimes referred to as "stove-pipe.") From multiple experiences such as the one described in Grenada, the DoD has learned that this vertical design paradigm leads to massive duplication of subcomponents and incompatibility between systems. Such a paradigm also interferes with, rather than facilitates, joint military operations. In light of the current downsizing in military force structures, coupled with the dwindling military research and development funds, the DoD must develop and field systems that are integrated both vertically and horizontally. All new automated systems must share their information across a wide spectrum of military applications that support interoperability requirements. These modified design environments are also essential for the required reduction in procurement costs.

The Department of Defense has made some progress in recent years towards adapting and embracing civilian research, development, and procurement practices. It now seeks to learn, adopt or adapt, as necessary, these corporate practices to meet unique military requirements, such as providing highly mobile communications and automated support in sparse communication environments. For example, utilizing a commercial mobile phone system as the baseline for development of a mobile military

communication platform makes sense; however, the military must often add redundancy to its networks. This redundancy is needed to permit communication to continue even when some communication lines are lost in battle. Other examples of considerations that military planners must address, but their commercial counterparts need not, include noisy Radio Frequency (RF) links, extreme bandwidth constraints, and security concerns.

Recent and ongoing developments in the area of information availability and accessibility within large, widely distributed, commercial corporations have direct applications within military applications. With high mobility and global-reach requirements, the military must develop and field automated systems that integrate all available and accessible information. The exploration and possible use of emerging technologies, such as **Asynchronous Transfer Mode (ATM)** [Ref. 1], for high-speed data communications are essential to the success of future military systems.

In one such effort, personnel at the Naval Command Control and Ocean Surveillance Center, Research, Development, Training, and Evaluation Division (NRaD), under the direction of Dr. Clifford J. Warner, are examining the use of ATM to satisfy the requirements for the transmission of multimedia data over the Navy's Joint Maritime Communication System [Ref. 2]. They are focusing on three areas:

- Actively participating in the Internet Engineering Task Force's efforts to standardize an approach to reliable multicast.
- Integrating Internet Protocol (IP) Quality of Service (QoS) guarantees with ATM QoS to ensure QoS support through multiple layers of the network, including the subnetwork level.
- Actively testing ATM equipment in their labs to identify the ability of ATM switches to:
    1. support multimedia applications,
    2. support IP and ATM QoS guarantees,
    3. interoperate with different ATM switches,
    4. support efficient multicast protocols, and

5. integrate with legacy systems.

These extensive efforts demonstrate a recognition by the military science and technology experts that ATM is a technology that might be exploitable in future Command and Control, Communication, Computer and Intelligence (C4I) systems.

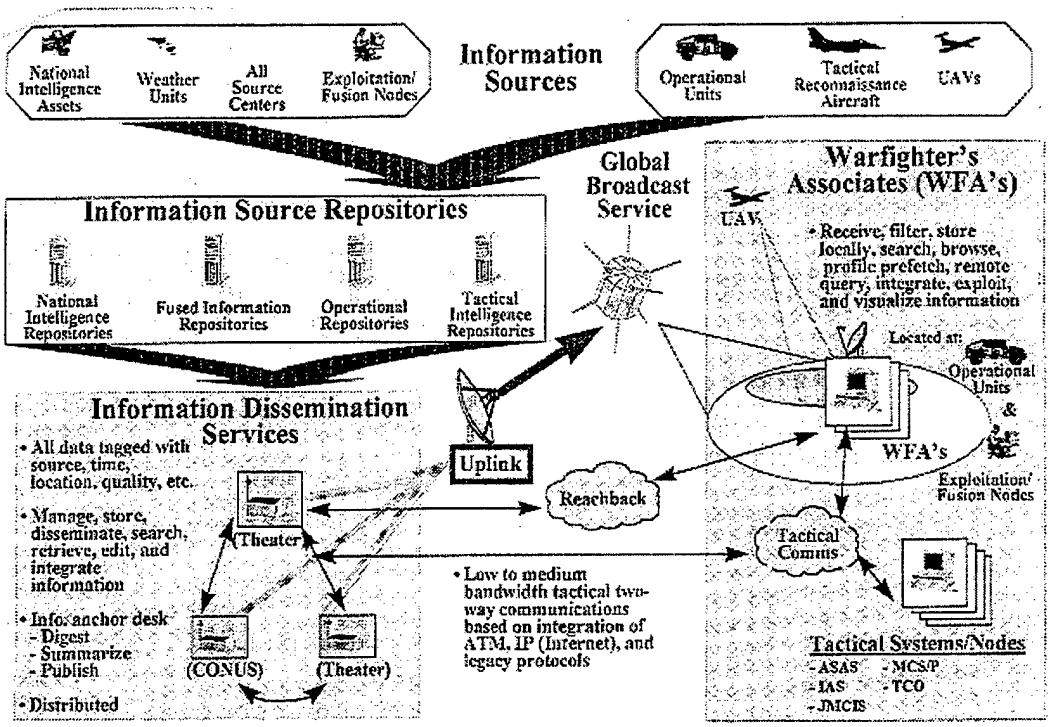


Figure 1. BADD Overview (From [Ref. 3]).

Figure 1 displays the **Battlefield Awareness and Data Dissemination (BADD)** System [Ref. 3]. The BADD system is an ATM-based communication system that has the primary goal of integrating all available information to **digitize the battlefield** and to ensure that the local battlefield commander maintains information dominance over opposing forces. The information needed to digitize the battlefield may be available either within the local operating forces environment or may be stored in the vast electronic libraries maintained by diverse organizations within the continental United States. These electronic libraries include the Defense Intelli-

gence Agency (DIA), the Defense Mapping Agency (DMA), and Cable Network News (CNN). The BADD program, under the direction of the **Defense Advanced Research and Project Agency (DARPA)**, attempts to integrate voice, data, video, and imagery in this single system.

Fundamental to the success of the BADD program is the rapid access and transfer of information to and from the battlefield. This transfer of information must include high-speed network connectivity over vast distances. Likewise, the program must include plans for scheduling retransmission of data that is lost due to network failures, which are likely in the military environment. Additionally, the military does not want to lose any important data. This requires the rescheduling of lower priority data when it is preempted by higher priority data.

This thesis investigates, through simulation, several problems resulting from scaling of the BADD project from its current prototype size to its eventually envisioned use. In particular, we use **OPtimized Network Engineering Tools (OPNET)**, a commercial software product from MIL3, to simulate BADD both in its current implementation and in planned future implementations [Ref. 4]. Our major work concentrates in the areas of scheduling algorithms for ATM virtual channels on the BADD satellite network link.

B. MOTIVATION

The motivation for our research stems from the need to ensure that the American forces and our allies remain the most informed forces on the battlefield. To accomplish this goal, the warfighter must immediately obtain all of the most current, relevant data. As the battlefield becomes digitized, the amount of operational, intelligence, and logistical information that may be of interest to the commanders is overloading the capacities of the existing networks. BADD examines new solutions that will permit the timely distribution of relevant information to even the lowest echelons.

Broadcast is an efficient means of getting information out to a wide audience. Broadcast information has been proven successful for the military with the Navy's Tactical Receive Applications Network and the Air Force's Tactical Information Broadcast System Network. Most recently, the **Bosnia Command and Control Augmentation (BC2A) Initiative**, supporting forces in the former Yugoslavia, has proven the usefulness of broadcasting heterogeneous sets of data. However, broadcasting all information to everyone will no longer be an option because there is simply too much information. Instead, we need to ensure that each of the warfighters has the information that is most relevant to them and that the information is as current as possible. To meet this need requires the use of a multicast protocol. The BADD program plans to use the **Global Broadcast Service (GBS)**, multiple static virtual ATM channels, and smart filtering to implement multicast of heterogeneous data to satisfy the demands of the forward deployed warfighter. One problem that the BADD designers have not yet tackled is that of scheduling the data to be broadcast over the multiple static virtual ATM channels in such a way so as to maximize the probability of the high priority data reaching the warfighter. The research for this thesis concentrates in that area, borrowing algorithms from SmartNet [Ref. 5], a framework for scheduling computation in a high performance, heterogeneous computing network.

C. METHODOLOGY

We first construct an OPNET simulation for the BADD network using its current configuration as a baseline model for further analysis. Then, we will run simulations using two different scheduling algorithms for scheduling data to be broadcast over the static virtual ATM channels. (The BADD architecture will hardwire these channels into the satellite uplink switch.) The first baseline simulation will run with First-In, First Out (FIFO) scheduling. The second simulation will incorporate intelligent scheduling with algorithms used in **SmartNet**, a United States Navy developed scheduler. We will then run multiple simulations to measure the maximum throughput

of the network with the different schedulers. Finally, we will perform analysis on our results in order to determine the utility of intelligent schedulers for these networks.

D. ORGANIZATION

The thesis is divided into seven chapters. Chapter II reviews communication fundamentals including TCP/IP and ATM protocols. Chapter III provides an overview of the BADD program and the general architecture of the network. Chapter IV elaborates on SmartNet's Intelligent Scheduling and the past successes of these algorithms. Chapters V and VI provide background information on the simulation software we are using and elaborates on the design of our simulation. Chapter VII presents the results of our simulation, provides the analysis of our simulation, makes recommendations for future work, and presents our conclusions. The appendices provide explanations of technical terms, computer code, scripts of simulations, and a list of acronyms used throughout this document.

II. COMMUNICATION FUNDAMENTALS

A. OVERVIEW

This chapter reviews some computer network basics. In particular, it concentrates on basic computer communication, basic data characteristics, basic **Transmission Control Protocol**(TCP), **User Datagram Protocol**(UDP), and **Asynchronous Transfer Mode**(ATM) services. If the reader is already familiar with computer networks, he or she can skip this chapter.

B. BASIC COMPUTER COMMUNICATION

Basic computer communication consists of three distinct components: a **source** (or sender), a **destination** (or receiver), and a **path** (or communication network) between the source and destination. Based on the complexity of the computer communication, either the source, destination, or the communication network can add header information to ensure reliability in the communication. The International Standards Organization (ISO) developed its Open Systems Interconnection (OSI) model for computer communication (see Table I) to isolate various functionalities within different layers. By grouping the communication functions into layers, communication services at a particular layer need only focus on the needs of that specific layer, using the services of the underlying layers. They need not be concerned with the implementation of the layers below.

C. DATA CHARACTERISTICS

The three basic parameters that describe most communication services are **time transparency**, **bit rate**, and **connection mode** [Ref. 1]. In the following subsections we will elaborate on each of these parameters.

Layers	Services
Layer 7 Applications	Provides access to users and provides distributed information services.
Layer 6 Presentation	Provides independence to applications using different data representations.
Layer 5 Session	Provides control structures for communication between different cooperating applications.
Layer 4 Transport	Provides reliable transfer of data between endpoints; provides end-to-end error recovery and flow control.
Layer 3 Network	Provides upper layers with independence from data transmission; responsible for managing communication connections.
Layer 2 Data Link	Provides reliable transfer of information across the physical link; responsible for synchronization, error control and flow control.
Layer 1 Physical	Transfers bits over a physical medium at the level of electrical signals.

Table I. Open Systems Interconnection (OSI) Model for Computer Communications (Modified From [Ref. 6]).

1. Time Transparency

Time transparency refers to the relationship, in time, between occurrences of events at both the source and the destination. Time transparency is formally defined as the near absence of **time delay** and **time jitter** [Ref. 1]. Time delay is the difference between the time the sender transmits the information and the time the information arrives at the receiver. Time jitter occurs when different parts of a transmission arrive at the receiver with different time delays. Network delay and jitter are primarily the result of two factors: propagation delay and processing delay.

Propagation delay is the physical delay of the medium; that is, the distance (d) that the signal must travel from the source to the destination, divided by the transmission speed of the physical medium. The transmission speed is bounded by the speed of light in a vacuum and ($\frac{2c}{3}$) in fiber optic cable where, c is the speed of light in meters per second. For example, the propagation delay in a one kilometer

fiber optic cable is given as

$$delay = \frac{1000}{\frac{2*(3*10^8)}{3}} = 0.00001 second.$$

Processing delay is the sum of the times required for the network hardware and network software to process the message at each node within the network.

Some communication services demand that the time required to complete a transmission be bounded. One example of a such a requirement is that time delay in voice data transmission must not exceed 25ms. When voice data time delays exceed this threshold, the receiver will notice "choppiness" in the conversation. Services that require a low bound on time delay are called real-time services.

2. Bit rate

There are four basic classes of bit rate services; constant bit rate, variable bit rate, available bit rate, and unspecified bit rate [Ref. 7].

- Constant bit rate is analogous to the bottles on a conveyor belt in a bottling plant. The bottles move through the process at a constant rate. Constant bit rate communication service places bits on the transmission media at a constant rate and takes them off the media at the same rate. Again, using the transmission of voice data as an example, the bit rate is constant at 64kbps for digital pulse code modulated (PCM) voice data. The input voice signal is sampled 8000 times per second and each sample is represented by 8 bits; each 8-bit sample is produced at this rate to give a constant 64kbps bit rate.

- Variable bit rate is analogous to train cars traveling on a railway. All cars are on the same track traveling at the same speed, however, the train cars have different sizes. Some cars are large size carrying two levels of automobiles, some are medium size carrying products in a box car, and some are small size like empty flatcars. If we view a railway tunnel as the size of a communications link, we can see that at times the tunnel's area is filled when large cars come through and is almost empty when flatcars come through. Video-teleconferencing is an example of a variable bit rate service. Using current video compression schemes. a base frame is sent, followed by a series of smaller frames containing the differences between the current frame and the base frame [Ref. 7].

- Available bit rate services is the service primarily used to support the World Wide Web. Due to financial constraints, companies cannot afford to install

communication services that will support their peak communication requirements. They instead decide upon some minimum communication capacity that must be obtained and then live with delays when peak data loads are applied. Because the communication service cannot support sustained peak load, this service must support congestion control to prevent network congestion that might lead to network failure.

- Unspecified bit rate has no congestion control and does nothing to ensure delivery. With this service, data is placed onto the network and, with luck, arrives at the destination. If congestion occurs within the network, data may be randomly discarded without notifying the user. File transfer and email are possible users of this type of service because applications have no constraints on time of delivery and they maintain their own, higher order, error control mechanisms.

3. Connection mode

A particular service may be **connection-oriented** or **connectionless** [Ref. 7]. In a connection-oriented service, a complete connection from sender to receiver is established before transmitting any data. An example of this type of service is the original telephone circuit switching system. Under the telephone system, a user picks up the phone and dials a telephone number. The telephone system establishes a dedicated circuit between the two users. This circuit is only used by these two customers until one customer hangs up the telephone, thereby releasing the connection. One might visualize this service as a tube, where one user at a time pours information in at one end and the other user takes the information out. The information will always arrive in the same order in which it was sent.

The other common connection mode is called connectionless [Ref. 7]. With this service, the user assigns a destination address to each message and places the message into the network. The user has no input and no knowledge about the route the message will take to its destination address. Furthermore, the user has no guarantee that two messages sent by the same user to the same destination will arrive at that destination in the same order in which they are sent. An example of a connectionless service is the postal system.

D. COMMUNICATION SERVICES AND PROTOCOLS

Now that we have defined basic computer communications and data characteristics, we begin to describe some common communication architectures which are used within the BADD system. As will be depicted in Chapter III, BADD is an integrated system of diverse communication services; this section will discuss each service with a focus on ATM.

1. TCP Data Transport Protocol

TCP resides at the transport layer (layer 4 in the OSI model), and is designed to work on top of an unreliable network [Ref. 7]. This connection-oriented protocol improves the reliability of communication by adding sequencing, acknowledgments, flow control, and congestion control mechanisms. Due to its complexity, TCP often becomes a major bottleneck in high-speed communications.

Figure 2 depicts the packet format for a TCP packet. The standard data field for a TCP packet is 1500 bytes long, with a maximum length of 64 kbits.

The fields within the TCP packet are as follows:

- **Source Port.** This is the sender's port for its application.
- **Destination Port.** This is the receiver's port for its application.
- **Sequence Number.** This field contains the sequence number of the first data byte in this TCP segment.
- **Acknowledgment Number.** Used for acknowledgment messages, this fields specifies the sequence number of the next data byte TCP expects to receive.
- **TCP HL.** TCP HL stands for TCP Header Length and specifies the total number of bytes contained in the header.
- **URG Flag.** A bit that, when set, tells the protocol to use the value in the Urgent Pointer.
- **ACK Flag.** A bit that, when set, alerts the protocol that the acknowledgment number is valid.
- **PSH Flag.** This bit, when set, activates the Push Function. The Push function pushes this packet up to the application layer even if the buffer is not yet full.

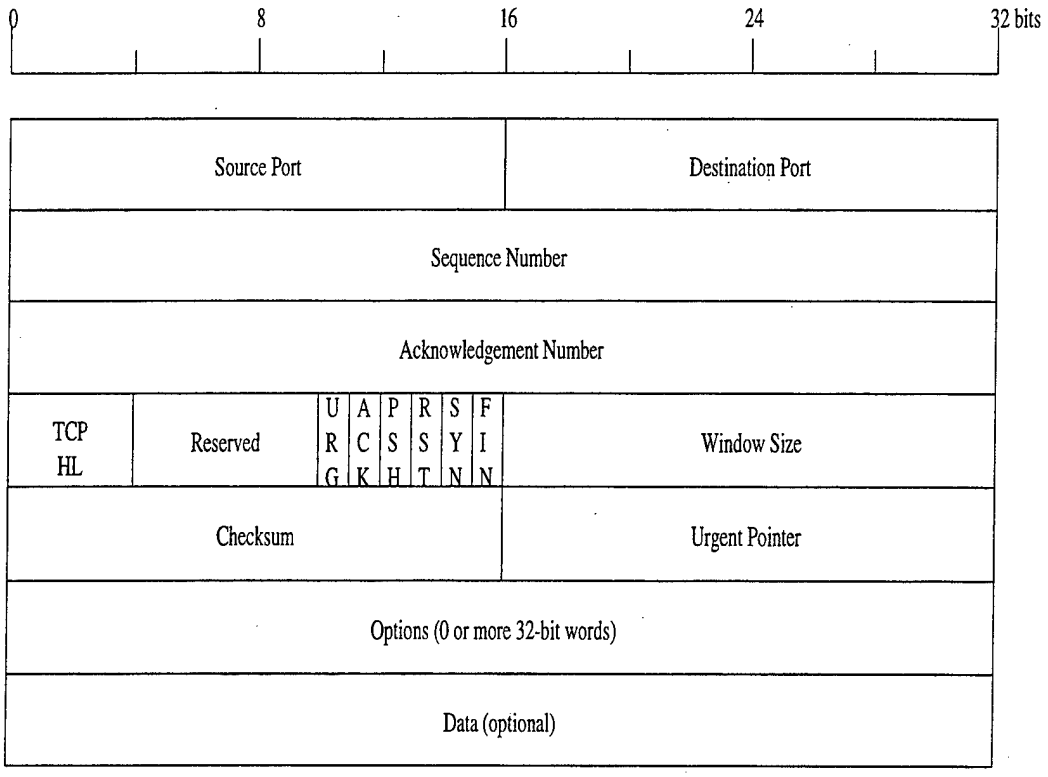


Figure 2. TCP Packet Format (From [Ref. 7]).

- **RST Flag.** This bit is set when either side detects problems with the connection and the connection must be reset.
- **SYN Flag.** A bit marking this packet as a request to establish a connection.
- **FIN Flag.** This bit is set when either the sender or receiver finishes with a connection.
- **Window Size.** Used by the sliding window flow control mechanism.
- **Checksum.** An error detection code for the header and data.
- **Urgent Pointer.** A pointer to the byte that immediately follows the urgent data.

2. UDP Data Transport Protocol

UDP, in contrast to TCP, provides a connectionless service for application level procedures [Ref. 7]. It also differs from TCP in its complexity and support for reliable communications. UDP is considered unreliable because it provides no mechanisms for

acknowledging receipt of data and no protection against duplicate receptions. UDP is often used when speed is more important and end-to-end reliability mechanisms are incorporated in the layer that sits above the UDP layer. Figure 3 depicts the packet format for the UDP packet. Like the TCP header, the maximum packet size for UDP packets is approximately 64 kbits.

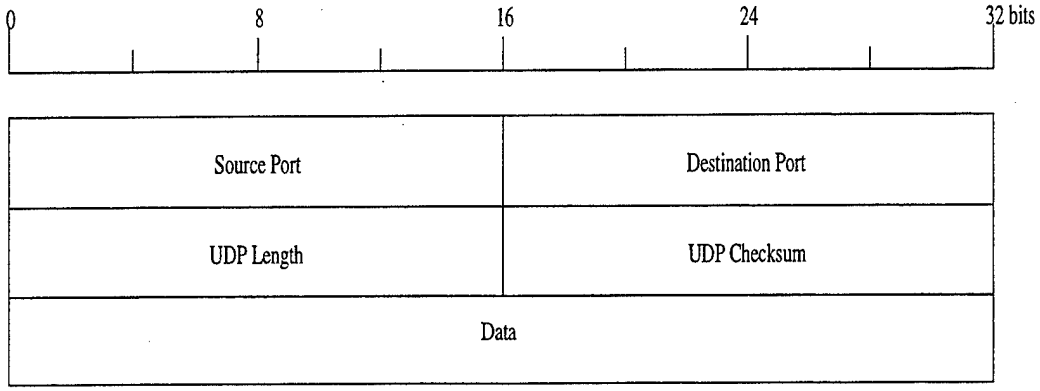


Figure 3. UDP Packet Format (From [Ref. 7]).

Each of the fields within the UDP packet are explained below.

- **Source Port.** This is the local port for the application, at the sender, using the UDP transport protocol.
- **Destination Port.** This is the local port for the application, at the receiver, using the TCP transport protocol.
- **UDP Length.** The length of the UDP header plus the data field.
- **Checksum.** The error detection code for the header and data.

3. ATM

a. ATM Fundamentals

Like TCP, ATM is a connection-oriented communication service. However, ATM is lower in the OSI model because ATM is also involved in routing from the source to the destination. Due to its multi-layer functionality, ATM does not "fit" cleanly into the OSI communications model. The conventional view is that the ATM protocol is a consolidation of the OSI Network and Data Link layers.

There are multiple sublayers within the ATM protocol and this section will discuss several of them in detail. Figure 4 shows the International Telecommunication Union Telecommunication Standardization Sector (ITU-T) protocol reference model for ATM. We provide this figure as a conceptual reference for the reader when we begin discussing the sublayers in detail.

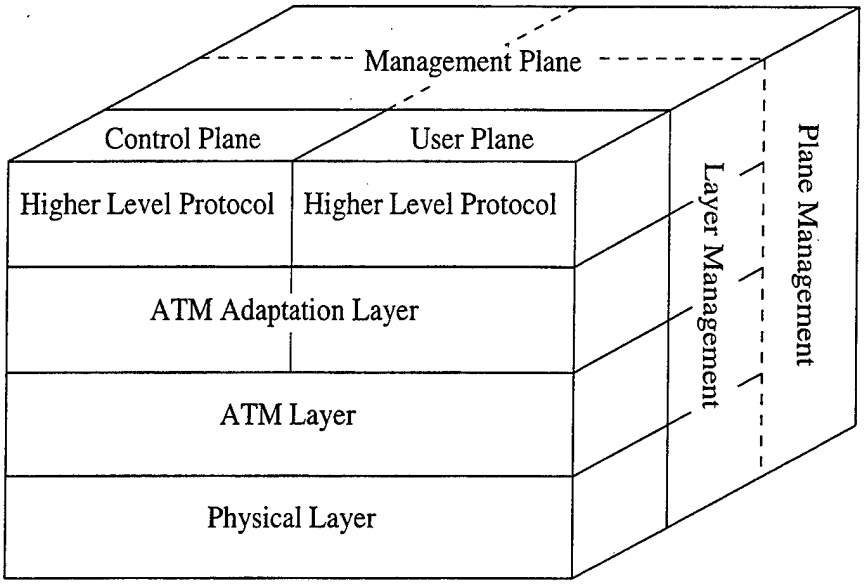


Figure 4. ITU-T ATM Protocol Reference Model (From [Ref. 1]).

The basic element of the ATM layer is a **Virtual Channel Connection (VCC)**. When data is passed to the ATM layer, a VCC is established between the source and destination. This VCC transports the user's datagrams, along with control signals that support the link and manage the overall network. After the user's datagram has been transmitted, the VCC is dismantled and its resources are returned. This process of connection and disconnection is extremely fast because much of it is incorporated in the ATM switch and network hardware.

The ATM protocol includes a **Virtual Path Connection (VPC)** which is responsible for bundling VCCs that have the same endpoint. To enhance overall ATM performance, all of the VCCs in the same VPC are treated as a single entity within the network. By reserving additional capacity on a VPC, in anticipation

of later VCCs, new VCCs can be established by simple, low overhead, control functions executed only at the endpoints. Figure 5 depicts the relationship between the physical route or cable, the virtual path (VPC) and the virtual channel (VCC).

Because ATM was primarily designed for use over fiber optic networks, and such networks are very reliable, the ATM protocol provides no mechanism for error control or error correction [Ref. 1]. These functions are left to the higher layer protocols. Additionally, ATM provides no mechanisms for acknowledgments; this function is also left to the higher layer protocols. Of course, there are occasional errors within fiber optic networks, but the probability is so low that the ATM protocol expects the higher layer to re-transmit an entire message if it detects an error.

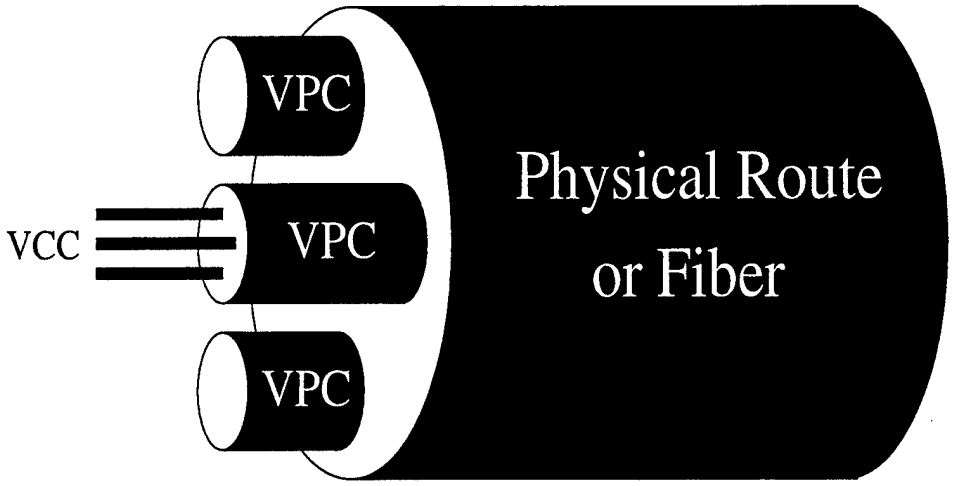


Figure 5. The relationship between Physical Route, Virtual Path Connection, and Virtual Channel Connection (Derived From [Ref. 6]).

Unlike TCP and UDP, ATM packets are all fixed length, and they are referred to as **cells**. Each ATM cell contains a 48-byte data field (payload) and a 5-byte header; it is extremely small compared to the average TCP or UDP packet. The International Telecommunication Union (ITU) standard defines two different packet header formats for ATM cells. One format, called User-Network Interface (UNI), is required at the boundary between the user host equipment and the ATM network (non-ATM

to ATM equipment interface). The second format, called Network-Network Interface (NNI) is used within an ATM network (ATM to ATM equipment interface). [Ref. 1]

Figure 6 depicts the two different ATM cell header formats.

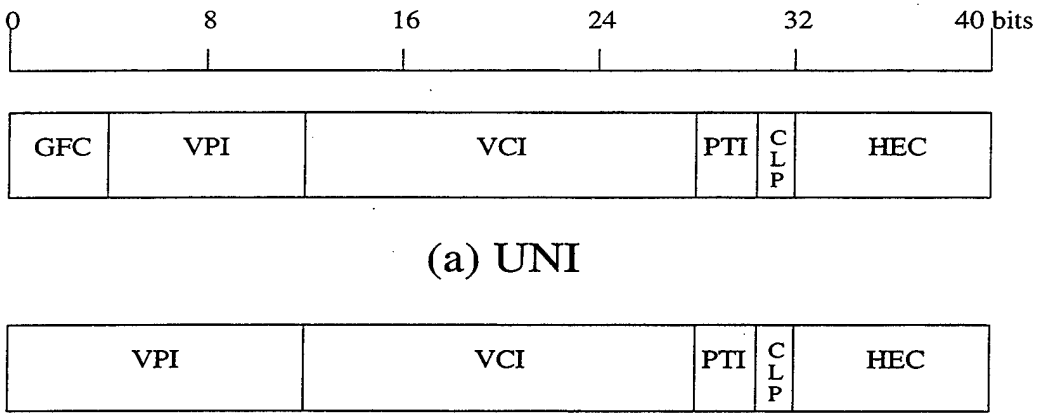


Figure 6. (a) ATM UNI cell header format. (b) ATM NNI cell header format (From [Ref. 7]).

Each of the fields within the ATM cell header are:

- **Generic Flow Control**. This field is only present in cells sent from a host to an ATM network and is currently not used. In the future, it could control cell flow or identify priority levels.
- **Virtual Path Identifier**. This field identifies the virtual path for this cell.
- **Virtual Channel Identifier**. The VCI identifies a particular virtual channel within the selected virtual path.
- **Payload Type Identifier**. ATM cells may carry user data or ATM management data. This field identifies the type of data contained within the cell.
- **Cell Loss Priority (CLP)**. The CLP bit flag distinguishes high and low priority data cells. A congested ATM network will first attempt to drop cells with CLP values of 1 before dropping those with a CLP values of 0.
- **Header Error Check**. The error detection code which is used to detect errors in the header only.

When analyzing the properties of the ATM protocol, we define a data stream as a group of cells that are sent from one user application to another. One of ATM's strengths is its ability to switch between data streams extremely efficiently because of the small, fixed length of ATM cells. When a higher priority data stream needs to interrupt a lower priority one, the higher priority stream must only wait for the completed transmission of a relatively small data segment.

The most common method for establishing priorities within an ATM network is by using a designator known as the **Class of Service**. Using the data characteristics previously defined, the ITU-T recommendation defines four classes of ATM service: Class A, Class B, Class C and Class D. Figure 7 illustrates the differences between each of these service classes.

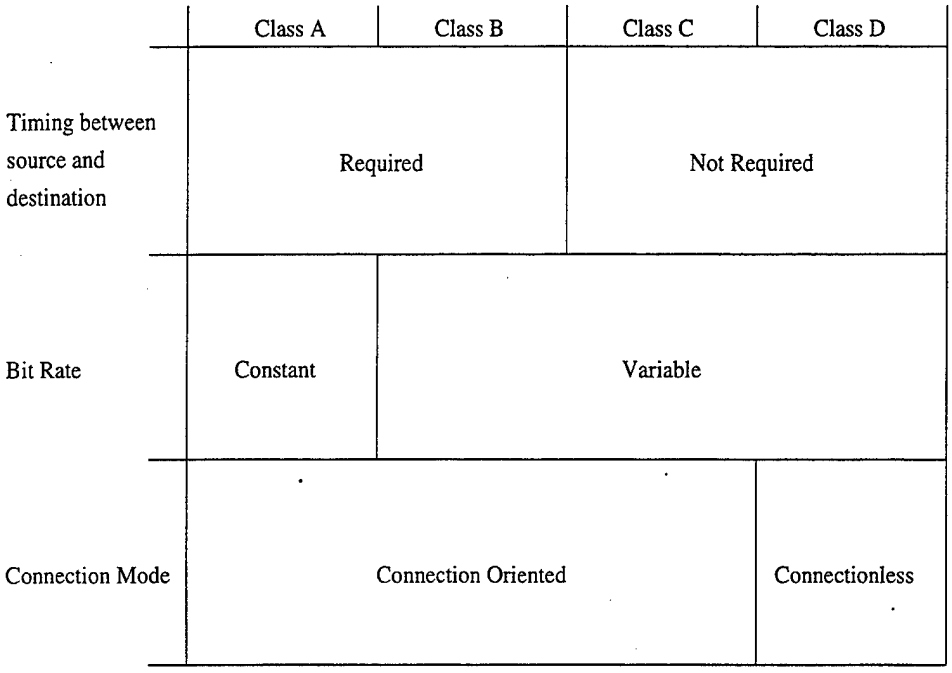


Figure 7. The differences between each of the ATM service classes (Modified From [Ref. 1]).

b. ATM's AAL Layer

The AAL sublayer sits just above the ATM sublayer and maps the user data, control data, and management data into the information field of one or more

ATM cells. Because there exists a vast difference between TCP/UDP packets and ATM cells, the ATM protocol contains a specialized sublayer to convert transport protocol packets to ATM cells. This sublayer is called the ATM Adaptation Layer (AAL).

The AAL layer consists of several different AAL types that are closely aligned to the different classes of ATM service. Table II defines each of the AAL types. This thesis is primarily focused on the AAL Type 5 service.

AAL Type	**Data Characteristics**
AAL 1	Constant Bit Rate services
AAL 2	Variable Bit Rate services
AAL 3/4	Data which is sensitive to loss but not to delay
AAL 5	Less sensitive to loss than AAL 3/4 but not sensitive to delay

Table II. ITU-T AAL type recommendations.

All of the AAL layers are subdivided into two sub-sublayers: the **Segmentation and Reassembly (SAR)** Sublayer and **Convergence Sublayer (CS)**. The CS layer is further subdivided into a **Service Specific Convergence Sublayer (SSCS)** and a **Common Part Convergence Sublayer (CPCS)**. The relationship between each of these layers is depicted in Figure 8.

We now address how all of the sublayers within the AAL layer function together to convert a large TCP/UDP packet into ATM cells for transport across a network. The SSCS is protocol-specific and may include message framing and error correction. The CPCS within AAL5 begins by padding the user data packet to make the entire message a multiple of 48 bytes. The CPCS then adds one byte for the User-to-User field, one byte for Common Part Indicator, two bytes for the user data length, and four bytes for error detection. The CPCS Protocol Data Unit is depicted in Table III.

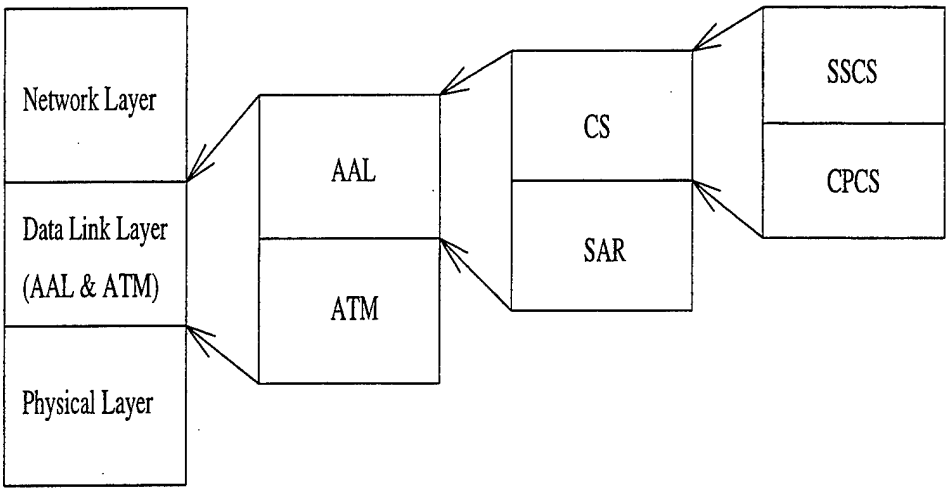


Figure 8. The relationship between ATM, AAL, SAR, CS, CPCS, and SSCS sublayers (Derived From [Ref. 4]).

User Data	Pad	UU	CPI	Length	CRC

Table III. CPCS-PDU format for AAL5

The Segmenting and Reassembly (SAR) layer is responsible for different functions depending on whether it is at the receiving or transmitting end of the network. At the transmission end of the network, the SAR segments large data packets into 48-byte segments. Each 48-byte segment is passed down to the ATM layer where they are placed into individual ATM cells. At the receiver end of the network, the SAR reassembles the 48-byte segment into the large data packet and forwards the large data packet to the receiver's application layer.

c. Transporting ATM cells across an ATM Network

After the AAL layer has converted calls from ATM and non-ATM into 48-byte segments, the cells are transported from source to the receiver. Before the cells can be transmitted across the physical layer, all ATM switches between the source and destination must establish a connection. This section describes the process for connection setup when a physical route exists from source to receiver and a special virtual circuit exists for signaling within that ATM network.

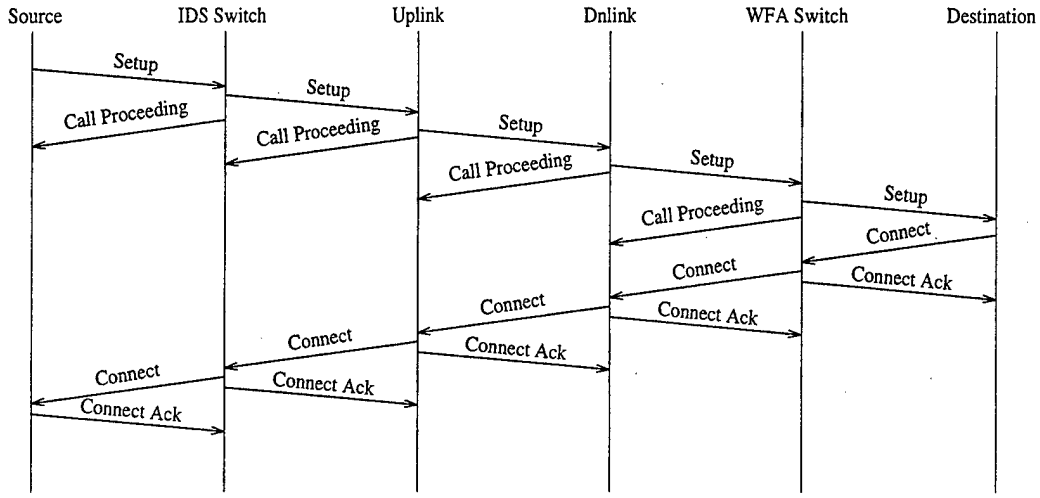
Virtual circuits are established by using the special signaling circuit and the six message types defined in Table IV [Ref. 7]. A signaling message may be sent by a host (source or receiver) or by switches within the network. A **call request** is a signal from a higher level application layer to the AAL layer indicating that data needs to be sent across the network. A call request begins when the source host sends a SETUP message over the signaling circuit; this process is depicted in Figure 9. The first switch in the network then sends a SETUP message to the next switch in the network and sends a CALL PROCEEDING back to the source. As the SETUP message works its way through the network, each switch forwards a SETUP message to the next switch and sends a CALL PROCEEDING back to the previous switch.

Message	Meaning at Host	Meaning within Network
SETUP	Request a circuit	Incoming call
CALL PROCEEDING	Network saw call	Attempted to establish call
CONNECT	Network accepts call	Requested call was accepted
CONNECT ACK	Connect message received	Call established
RELEASE	Terminating the call	Call terminated
RELEASE COMPLETE	Ack for RELEASE	Ack for RELEASE

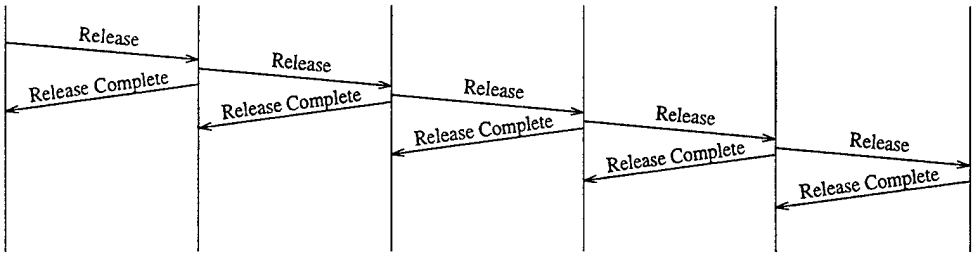
Table IV. ATM Messages used for connection establishment and release (From [Ref. 7]).

When the SETUP message finally reaches the destination ATM Switch, the destination transceiver responds with a CONNECT message if the destination accepts the call request. The CONNECT message works its way back to the source in a manner similar to the SETUP message, except in the reverse order. At each switch within the network, a CONNECT ACK message is sent to the previous switch. When the CONNECT message arrives at the source, the source also sends a CONNECT ACK message to the previous switch in the network.

Terminating a virtual circuit is simpler than establishing it. The source sends a RELEASE message to the first switch in the network. This message propagates through the network with a RELEASE ACK message sent at each hop along the route.



(a) Call Establishment Sequence



(b) Call Release Sequence

Figure 9. (a) The sequence of signaling messages for establishing an ATM virtual channel. (b) The sequence of signaling messages to release an ATM virtual channel (From [Ref. 7]).

III. OVERVIEW OF THE BADD PROGRAM

A. OVERVIEW

We first provide a general description of The **Battlefield Awareness and Data Dissemination (BADD)** system. Subsequent sections of this chapter provide a more detailed description of its major components [Ref. 8]. The BADD project is an **Advanced Concepts Technology Demonstration (ACTD)** sponsored by the **Defense Advanced Research Projects Agency (DARPA)**. BADD uses commercial off-the-shelf (COTS) systems to support the U.S. military's need for total battle awareness, that is, to provide information where it is needed, in the correct format, and in a timely and cost effective manner. The goal of BADD is to empower war-fighters, from the Task Force Commander through the Battalion Commander and below, with information. Such information will be delivered using advanced dissemination technologies. The BADD system consist of the following major components:

- the Information Dissemination Server (IDS),
- the Satellite Broadcast System,
- the Tactical System/Warfighter Associate (WFA), and
- the Reachback Link.

An overview of the interactions of these components is depicted in Figure 10 and is summarized below.

The **IDS** functions as the central repository for processing, storage, and retrieval of data. The **Satellite Broadcast System** forwards data from the IDS to multiple tactical users, via satellite communication. Data broadcasted over the satellite link is collected and stored at the tactical units by a suite of automation equipment called the**Warfighter Associate (WFA)**. The data is then made available to the warfighter's Local Area Network (LAN) via the Tactical Internet (TI). The **Reachback Link** allows tactical units to forward important data, such as location

information or unmanned aerial video, back to the IDS for timely dissemination to other tactical forces. In addition, it provides a means for the tactical units to request information from the IDS. Some units will not have a reachback link available to communicate to the IDS directly and will relay their request to the closest unit with reachback. These units will have the ability to "listen in" to broadcasts, filter out unwanted data, and save what they need. Next we will examine each of these components in greater detail.

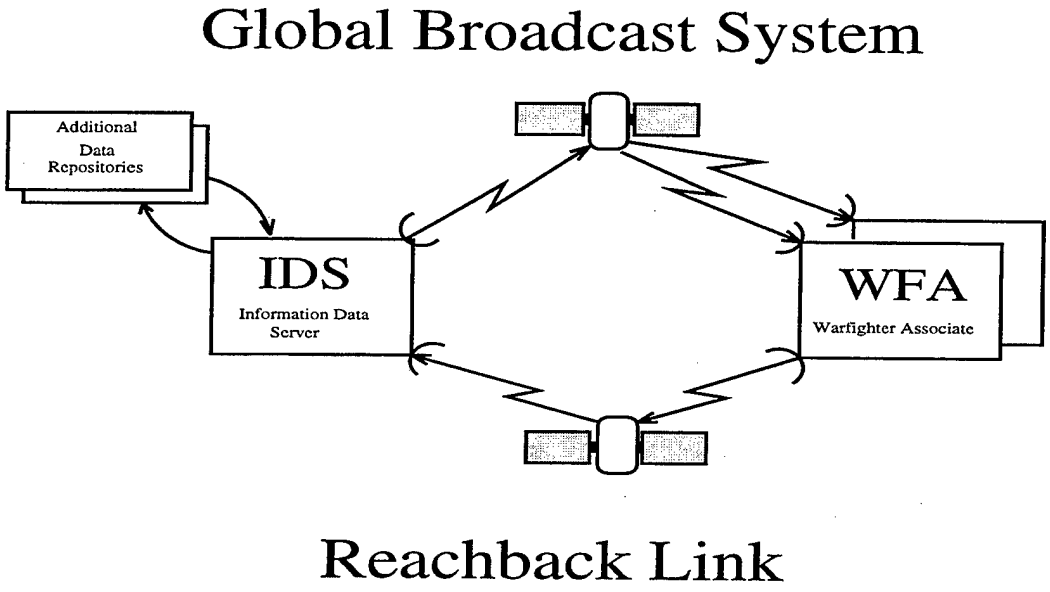


Figure 10. BADD Overview (From [Ref. 8]).

B. INFORMATION DISSEMINATION SERVER

The Information Dissemination Server (IDS) provides management, storage, and dissemination to the broadcast uplink facility. The IDS is located at DARPA's **Advanced High Performance Computing Applications (AHPCA)** lab facility. The IDS works in conjunction with the Warfighter Associate (WFA) to satisfy the information needs of the lower echelon commanders. The IDS receives and stores information from in-theater via the reachback links, and can access information from multiple resources, such as CNN, national intelligence sources, and the National

Weather Service, in the Washington, D.C. area. It forwards data to the field in two ways. The first is by broadcasting the data that is stored locally. The second way is by acting as an intermediate communication hub, forwarding data from either a data repository or the reachback link without storing the data locally.

The IDS has the functionality to integrate data from various resources and disseminate a more comprehensive view of the battlefield to the Battalion Commander in the field. One of the ways the IDS accomplishes this mission is through intelligently prepositioning information at the WFA for rapid retrieval. Such prepositioning is referred to as a **Push** operation. Another way the IDS satisfies the needs of the user is through a **Pull** operation. A pull operation is when the IDS forwards information to be broadcast in order to satisfy a query from the field over the reachback link.

Figure 11 depicts a high-level diagram of the flow of data between the major subcomponents of the IDS. We will elaborate on each of these subcomponents in the remainder of this section. [Ref. 8]

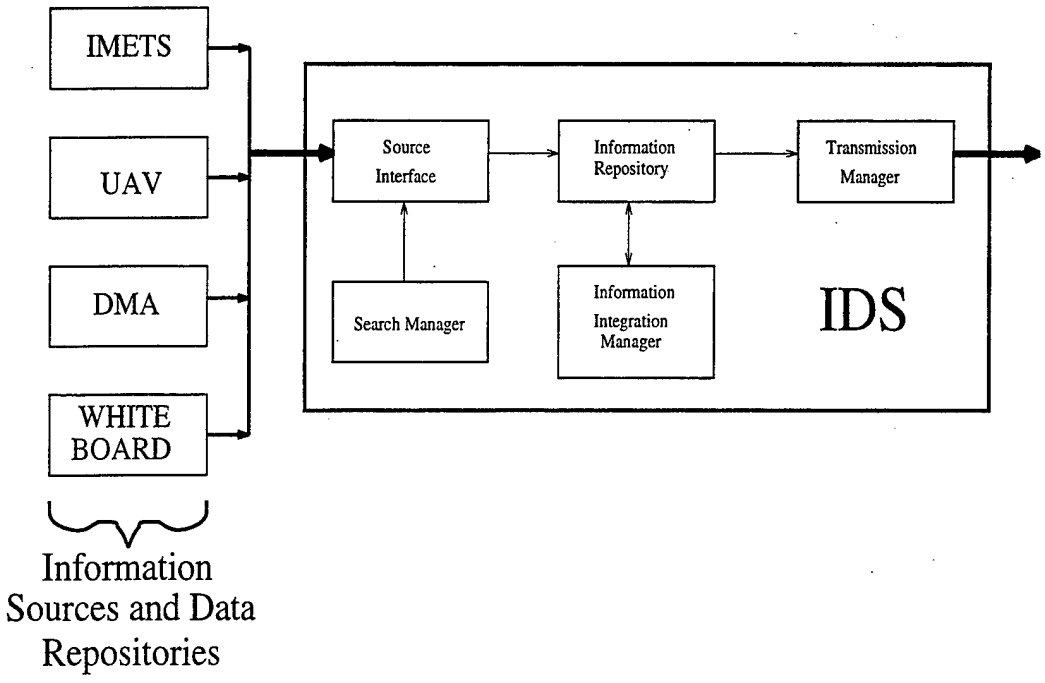


Figure 11. IDS Components (From [Ref. 3]).

1. Information Sources and Data Repositories

Figure 11 shows that there are multiple sources of information coming into the IDS. The sources we enumerate are not inclusive and are meant to show the variety of data types that can be made available through the BADD architecture. Table V describes each of the sources depicted in that figure. [Ref. 3]

Source	Description
IMETS	Integrated Meteorological System weather data from the division Tactical Operations Center (TOC).
UAV	Unmanned Aerial Vehicle broadcast from tactical units. Forwarded live for wider dissemination.
WHITE BOARD	Commander's Whiteboard. Provides graph for the commander to use to elaborate on orders/instructions.
DMA	Defense Mapping Agency map information.

Table V. IDS Information Sources (From [Ref. 3]).

2. Source Interface

The **Source Interface** integrates the data that it receives from multiple data sources, including the reachback link. To accomplish this task it uses FORE ATM switches that allow inputs from both ATM sources and CISCO Routers. CISCO Routers allow input from TCP/IP/UDP sources.

3. Search Manager

The **Search Manager** retrieves information from the remote repositories in order to satisfy Priority Information Requirements (PIR) from the tactical commander. To accomplish this task it maintains directories indicating which resources are available from each of the distributed repositories.

4. Information Dissemination Repository

The **Information Dissemination Repository** provides a very large (tera-byte to peda-byte) on-line storage capability and is optimized to service queries for

multimedia data. This subcomponent supports the use of distributed heterogeneous archives. These archives can be geographically separated, can be forwarded to the IDS by different communication protocols (TCP/IP/UDP), and can contain data ranging from small text files to very large video files.

5. Information Integration Manager

The **Information Integration Manager** correlates data from the Information Dissemination Repository in order to integrate and validate data from different sources. It also searches for redundancies in the data in order to eliminate them and provide both a more efficient data transfer and a more user-friendly interface.

6. Transmission Manager

The **Transmission Manager** maximizes the probability of reliable delivery across simplex, error-prone satellite channels. It optimizes the dissemination to tactical commanders by utilizing algorithms and heuristics that send the highest priority data during peak periods and smartly push data during low utilization periods.

Determining when to push data is referred to as the **Data Staging Problem**. H. J. Siegel and some of his Ph.D. students at the University of Purdue are working to resolve this problem [Ref. 9]. Their efforts focus on determining the best data to cache, given:

- the priority of data and its perishablility,
- the dynamic nature of requests and network congestion, and
- limits on bandwidth and data storage.

They are examining the use of mathematical algorithms, such as Dijkstra's Algorithm, to improve the performance of BADD transmission management.

C. SATELLITE UPLINK AND BROADCAST SEGMENT

The **Satellite Uplink** facility is connected to the IDS via high speed links (T3) and is the insertion point for data into the broadcast link connecting the IDS

to the Warfighter Associate. This broadcast facility is responsible for providing efficient utilization of the link while ensuring that the priority scheme for transmission is not violated. The broadcast is transmitted over a geosynchronous Ku-Band satellite, providing a direct link from the uplink system to the battalions in the field. The broadcast has two channels, one for video and the other for data. The data channel is comprised of numerous data channels multiplexed together to form one wideband channel. Figure 12 depicts the interactions of the major subcomponents of the Broadcast Subsystem; each of these subcomponents is addressed in this section.

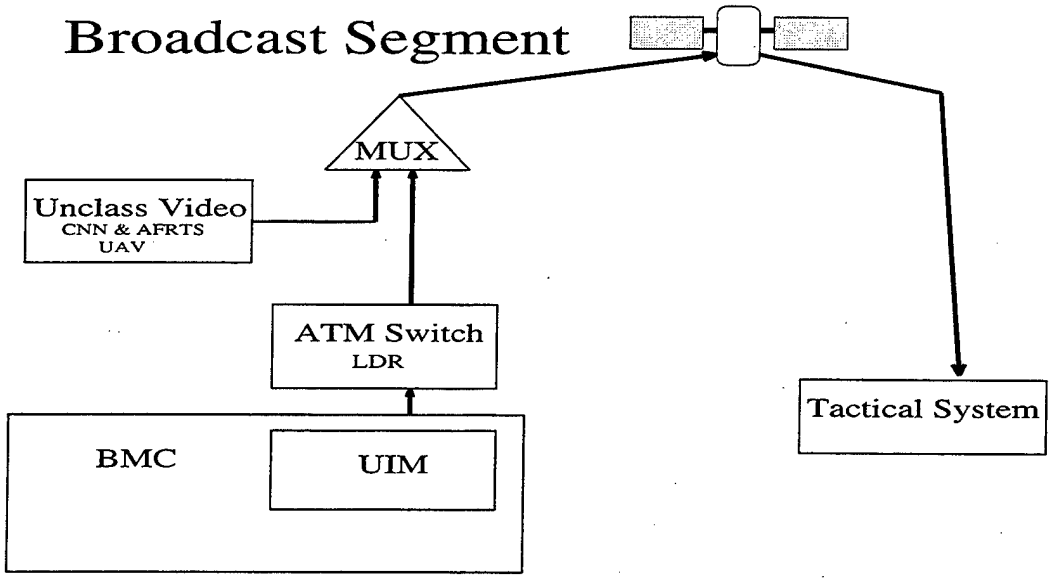


Figure 12. Broadcast Segment (Derived From [Ref. 10]).

1. Broadcast Management Center

The **Broadcast Management Center (BMC)** software is responsible for collecting all information to be broadcast to BADD tactical users. The data received at the BMC will include information requested by tactical users (warrior pull) and time-sensitive, intelligently-selected (smart push) information for broadcast to tactical users.

The BMC will decide answers to the following questions:

- What information needs to be broadcast?
- What priority is the data to be broadcast?
- What bandwidth is needed for the data?
- When should the data arrive at the Warfighter Associate?

If a message cannot be broadcast within the BMC-assigned parameters, the BMC must review the information and determine whether the information can be broadcast at a lower priority or whether the information should be discarded.

2. Uplink Information Manager

Within the BMC, the **Uplink Information Manager (UIM)** manages all information for broadcast through the ATM switch. The UIM is primarily concerned with maximizing throughput, given a priority level. Closely coordinating with the ATM switch to determine the dynamic ATM capacity, the UIM attempts to schedule all data transmissions to achieve data throughput of the highest priority messages. If the UIM cannot schedule the broadcast of a message within the assigned constraints, it will return the message to the Broadcast Manager for resolution.

3. Asynchronous Transfer Mode (ATM) Switch

The ATM switch currently used is the Integrated Systems Technology (IST) Low Data Rate (LDR) Model LDR-100S [Ref. 8]. It supports both ATM and non-ATM links, providing multimedia, video teleconferencing (VTC), mobile subscriber equipment (MSE), and tactical packet network (TPN) support. The system contains four special buffers to minimize the delay of the constant bit rate (CBR) traffic and allow for smoothing of peak data rates. It also contains a TAXI interface card that supports connections for one serial system and one parallel port. The previous chapter of this thesis described the ATM protocol in detail.

4. Unclassified Video

This unclassified video consists of CNN news broadcasts, Armed Forces Radio and Television broadcasts, unclassified Unmanned Aerial Vehicle broadcasts, and other unclassified video broadcasts. The current configuration allocates half of the total transmission capability to transmission of unclassified video.

5. Multiplexor (MUX)

The 30 Mbps **Multiplexor** will multiplex the unclass video and ATM data into a single data stream for broadcast transmission over the Global Broadcast System.

6. Global Broadcast System

Currently, the **Global Broadcast System (GBS)** uses a commercial Telestar 401 Ku-band satellite transponder, in geosynchronous orbit, that provides an effective transmission capacity of 23.6 Mbps. This single satellite in the GBS constellation is capable of providing coverage over the continental United States.

D. TACTICAL LEVEL SUBSYSTEM

Tactical units at the battalion level will have a **Receive Terminal** that consists of a satellite antenna dish, an amplifier/converter called a Low-Noise Block (LNB) converter, two Integrated Receiver Decoders (IRD) that provide the decoding of either a data or video signal stream, and an ATM switch that provides the first level of filtering, a KG-194A Decryption Device, and the Warfighter Associate. Figure 13 depicts a high-level diagram of the major subcomponents of the Tactical Level Subsystem. Each of these subcomponents is addressed in this section.

1. GBS Receive Equipment

The receive equipment consists of a 1.2 meter antenna dish connected into a Low Noise Block (LNB) converter. The LNB is a converter taking the Ku-Band (11.7-12.2 GHz) satellite transmission and converting it to L-Band (0.39-1.6 GHz). This segment also includes an Integrated Receiver Decoder (IRD) that converts the

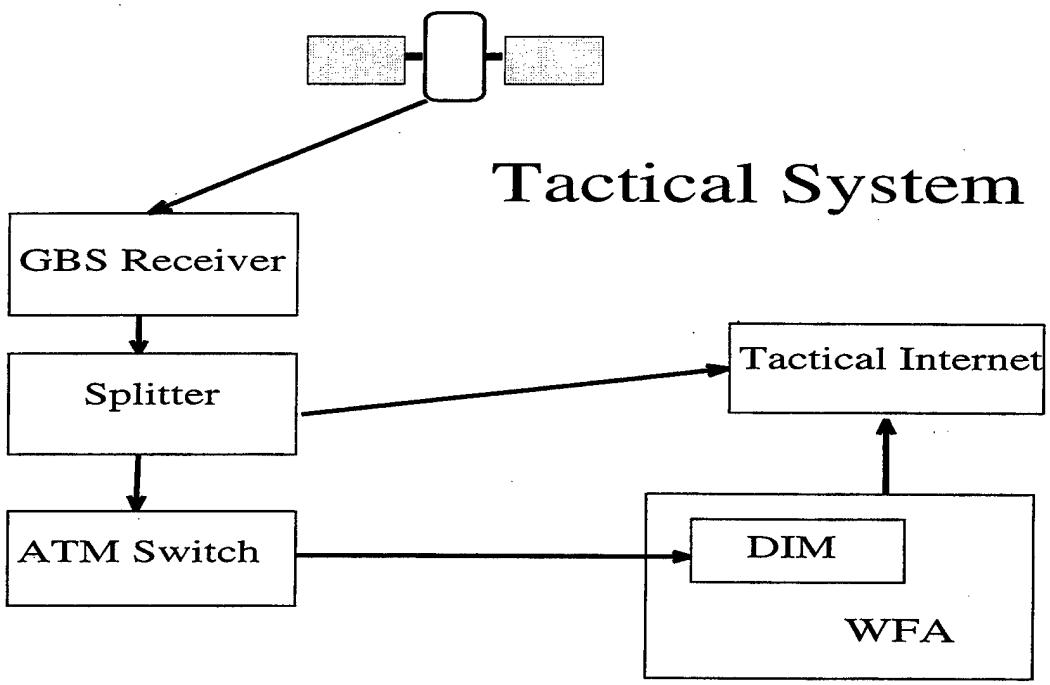


Figure 13. Tactical Segment (Derived From [Ref. 8]).

signal to a parallel data stream for forwarding to the ATM Switch.

2. Splitter

The **Splitter** accomplishes one simple function. It takes the non-ATM signal on the broadcast and splits it off for injection into the tactical internet, while allowing the ATM transmissions to pass through to the ATM Switch. Non-ATM signals include CNN, AFRTS and other commercial television broadcasts.

3. Downlink Information Manager (DIM)

The **Downlink Information Manager (DIM)** is the subcomponent of the Warfighter Associate that determines whether the information currently being broadcast is of interest to the tactical user. The WFA user customizes his environment based upon a "television guide-like" broadcast sent over the a predefined control channel on the satellite link displaying the scheduled broadcast. This function is critical because the Warfighter Associate only has limited local storage capabilities, and additionally, the tactical user does not want to be flooded with irrelevant data.

4. ATM Switch

The switch used is the same IST LDR-100s that is used at the uplink facility. It provides all of the same functions described above, but in support of reception of the ATM broadcast instead of its transmission.

5. Warfighter Associate (WFA)

The **Warfighter Associate** is the heart of the BADD system from the tactical commander's perspective. It provides his local data repository for quick access to the most important data. The WFA also provides information integration and battlefield functionality. The Warfighter Associate Workstation is an Ultra-SPARC unit responsible for filtering the incoming GBS/BADD data based upon specific profiles, areas of interest, and areas of operations. Workstations will vary in their configuration based on the service, but a typical workstation has 64 megabytes of random access memory (RAM) and 10.4 gigabytes (GB) of hard disk storage with 2 GB internal and 8.4 GB on external disks.

The WFA workstation provides the following functions:

- Stores data in a local repository,
- Provides users with applications and interfaces,
- Performs information integration operations,
- Provides a graphical display for battlefield visualization, and
- Provides access to other users on the tactical LAN.

6. Tactical Internet

The **Tactical Internet (TI)** is a subcomponent of the **Army Battle Command System (ABCS)** that focuses on providing support for battle command at the brigade level and below. The TI is focused on providing situational awareness to warfighters at lower echelons through the use of Applique devices. Applique devices that are comprised of: 1.) a computer host and software package; and 2.) a tactical

radio system that provides duplex information flow between a platform (i.e., tank, artillery, or aircraft) or an individual soldier and the division level of command. System Integration Vans (SIVs) plan and manage the use of the Tactical Internet and take inputs from the following systems:

- Enhanced Position Location Reporting System High Speed Integrated Circuit (EPLRS VHSIC),
- Surrogate Data Radio (SDR),
- Single Channel Ground and Airborne Radio System (SINCGARS) System Improvement Program (SIP),
- Mobile Subscriber Equipment (MSE),
- BADD, and
- Interfaces to Tactical Operations Center (TOC) Local Area Networks.

E. THE REACHBACK SUBSYSTEM

The **Reachback Link** provides the connectivity from the tactical unit to the IDS and has two different sublinks. A Trojan Spirit Satellite communication system provides an aggregate transmission rate of 1.544 Mbps from the Brigade TOC to the IDS LAN. This first system utilizes an unclassified ethernet-based sublink running the IP protocol. It requires a **Tactical End-to-End Encryption Device (TEED)** because it tunnels through a an operationally classified network. Unmanned aerial vehicle imagery, operational and intelligence overlays, and IMETS data are sent over this sublink. The latest testing results from an exercise in November 1996 required that the largest packet size transported from the Brigade WFA to the IDS was 1472 bytes. The average round trip time for a 64-byte packet, from the Brigade WFA through the reachback link and broadcast link back to the WFA, was 564 ms [Ref. 11].

The second system is ATM-enabled sublink and uses a KG-194A encryption device to provide secure data transmission. It operates at 768 Kbps and sends one-way video stream and one-way collaborative planning to disseminate the Brigade

Commander's Intent with real-time video, audio, and electronic whiteboarding. The round trip time for a 64-byte packet on this subnet was 808 ms. Jeffrey Carpenter and Jim Galanis, Electronics Engineers with the US Army Communications-Electronics Command (CECOM), recommended examining the effects of reducing the bit rate to 384 Kbps in order to provide the Ethernet sublink with more bandwidth, based on their November 1996 experience [Ref. 11].

Figure 14 depicts a high-level diagram of the major subcomponents of the Reachback Subsystem. Each of these subcomponents is addressed in this section.

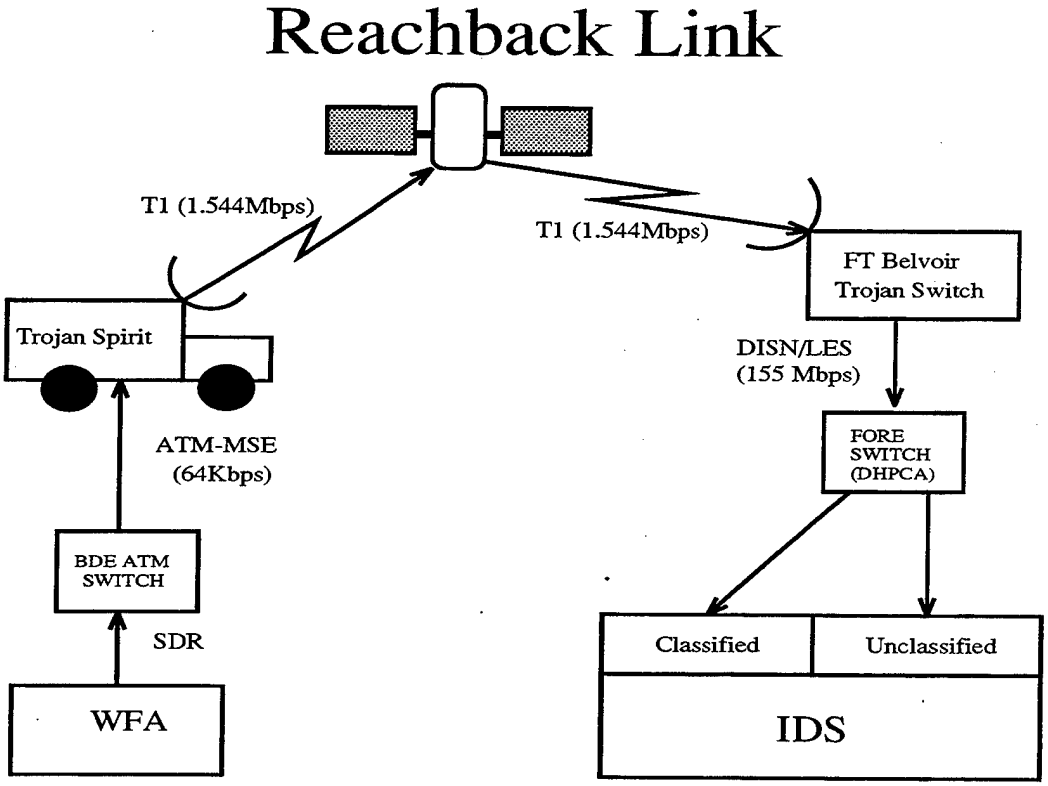


Figure 14. Reachback Components

1. Surrogate Digital Radio

The **Surrogate Digital Radio (SDR)** is a state-of-the-art digital radio that provides communication between the TOCs, the alternate TOCs, and command and control platforms [Ref. 8]. It has two major subcomponents; a secure packet radio

(SPR) and a wireless integrated services digital network interface unit (WIU). The SPR is a spread spectrum modulation radio that operates in the UHF range and has a data throughput capacity of 190 Kbps and embedded communications security. The WIU is an 80486 lightweight computer providing a synchronous serial communication card that interfaces with the SPR and the TOC's ethernet LAN.

2. Brigade ATM Switch

The BDE ATM Switch provides a tactical ATM backbone switching support for all tactical users. The switch provides connections for wideband fiber optics, synchronous optic network (SONET) radios, employed tactical digital radios, and digital transmission group (DTG) network interfaces. The LDR-100, which was described earlier, is once again the chosen hardware for this switch.

3. Trojan Spirit

The **Trojan Spirit** provides secure satellite voice and data communications from the forces in the field back to the IDS [Ref. 12]. It is a highly mobile system mounted on the back of a High Mobility Military Wheeled Vehicle (HMMWV) Shelter with a 2.4 meter, C through Ku-band, Satellite Antenna. It tows a 5.5 meter X-Band satellite communication antenna mounted on an accompanying trailer. It can provide 14 channels of digital voice or support digital data subscribers over a T1 link (1.544 Mbps).

4. Satellite System

The Trojan Spirit transmits its data to a variety of satellite platforms including INTELSAT, GSTAR, and PANAMSAT. Transmissions on these frequencies and other satellite-lettered bands are listed in Table VI.

5. Fort Belvoir Trojan Switch

The Fort Belvoir Trojan Switch receives data from the Trojan Spirit in the field and forwards data via Defense Information Systems Network Leading Edge Services (DISN LES).

Letter Design	Band	Freq Range (GHz)	Bandwidth (MHz)	Use
L	UHF	0.39-1.6	50	Commercial, Military
S	UHF/ SHF	1.65-5.2	90	Tracking, Telemetry and Command
C	SHF	3.9-6.2	500	Commercial
X	SHF	5.2-10.9	500	Military
Ku	SHF	10.9-17.25	500	Commercial
Ka	SHF	17.5-30	2500 1000	Commercial Military

Table VI. Satellite Lettered Band.

6. AHPCA ATM Switch

The AHPCA ATM Switch in Arlington, Virginia processes the signals from the Fort Belvoir switch and forwards them to the proper location within the IDS. The FORE ASX-200 BX ATM switch provides the capabilities to function as a LAN backbone [Ref. 13]. These switches provide advanced reliability and fault tolerance, and can support up to 24 clients, servers, and LAN devices such as hubs, routers or LAN switches. They execute ForeThought Internetworking Software that provides connection management features such as on-demand switched virtual circuits (SVCs), both User Network Interface (UNI) and Network Network Interface (NNI), and transparent support for existing LAN applications that use IP-over-ATM and LAN emulation.

7. Summary

We have now reviewed all current components and subcomponents of the BADD Program. We want to emphasize that this is a dynamic experimental architecture with components being added and removed. Our description of the system should be viewed as a single instantiation, as of April 21, 1997. Individual components and even entire subsystems may change. In the next chapter we review some of the scheduling algorithms that we considered to integrate into the BADD Architecture.

IV. INTELLIGENT SCHEDULING

A. INTRODUCTION

In Chapter II we described ATM's signaling procedures for virtual channel setup and tear-down. In this chapter we will examine the application of some commonly used scheduling heuristics that we expect will improve the throughput and fairness of the BADD network's ATM configuration. Some of these heuristics are used in the U.S. Navy's SmartNet Scheduler [Ref. 14].

In a standard ATM implementation, the request to establish a channel contains a description of the requirements, which includes the requested Quality of Service (QOS) and the Peak Cell Rate (PCR). As the setup request travels through the network, resources are allocated to support the virtual channel. Normally, these virtual channels are created as needed and then destroyed after the data transmission completes. This coordination requires duplex communication between the sender and the receiver.

Within the BADD architecture, the GBS satellite link provides only simplex data communication between the IDS and WFA ground stations. The BADD architecture therefore relies on static virtual channels. A static number of virtual channels are defined at the uplink and downlink ATM switches and these channels are statically allocated a particular bandwidth as well as other attributes, such as guaranteed latencies. Future implementations could define 1024 static virtual channels within the BADD ATM network [Ref. 15].

With static channels defined, BADD must employ some form of scheduling to assign a transmission request to a specific channel. This chapter reviews the basic scheduling problem and possible solutions.

B. SCHEDULING PROBLEM

The problem of determining the schedule that optimizes throughput for executing a set of heterogeneous tasks using a fixed set of heterogeneous resources is one of the most difficult and challenging problems in computer science. This general heterogeneous scheduling problem and other difficult problems comprise a set of problems known as **NP-Complete** problems. Most theoretical computer scientists believe the NP-Complete set of problems are intractable [Ref. 16]. For such problems, heuristics with polynomial runtimes are applied to attempt to obtain a near-optimal solution.

In the next sections we will examine some heuristic-based scheduling algorithms we considered implementing in BADD. We provide the psuedo-code for each algorithm, an example of its execution to enhance understanding, and a discussion of its strengths and weaknesses.

C. HEURISTIC ALGORITHMS

In this section we describe some heuristic algorithms and explain both their advantages and disadvantages when applied to the general heterogeneous scheduling problem. Throughout the section we use the term **call** when addressing a pending request for a virtual channel. We use this term to remain consistent with the terminology that is used when we describe the code for our OPNET implementation of BADD. In ATM terminology, a call refers to a session between a source and a destination. A call can contain one or many **messages** inside it. A message is defined as a group of packets that belong to one specific communication application (i.e., an email message or a database view). For this thesis, we assume that a call contains only one message.

1. First In First Out

The First In First Out (FIFO) algorithm is the simplest algorithm. It assumes that every channel can be used for every transmission. The scheduler enqueues all pending calls onto a wait queue and schedules the call at the head of the queue on the next available channel.

a. FIFO Algorithm Specification

The following is a general algorithm for simulating FIFO Scheduling. This algorithm can be modified if some calls are not suited to some channels.

```
1.  Place all jobs to be scheduled in the time-ordered queue,
Job_Queue, and set channel_avail_time[channel_number] = 0 for all
channels

2. While (Call_Queue not empty)

    3.  Remove first call in the Call_Queue.

    4.  Compute earliest time at which a channel will finish
        sending its current call. Store the index of that channel
        in schedule_index.

    5.  Schedule call that was removed from Call_Queue for
        channel[schedule_index].

    6.  Update channel_avail_time[schedule_index] by adding to
        it the amount of time needed to complete the removed
        call.

7. End While
```

b. Examples

Figure 15 shows an example of the steps for the simulation of the FIFO algorithm. There are three VCs, and currently there are three pending calls. In step (a), the call scheduling matrix is created, which contains the required time to complete each call when scheduled on each of the virtual channels (VC). We depict the completion times for previously scheduled calls on each of the virtual channels using a timeline. In this example, virtual channels 1 through 3 will finish at times 11, 15, and 60, respectively. In step (b), we select VC1 to service call 1 because it will be available at the earliest time, $t = 11$. (Notice that the values calculated in the call scheduling matrix are not used when selecting a channel. The matrix is only used to update the timeline values.) Next we increment the timeline to reflect that call 1

has been scheduled on VC1. In step(c), we repeat the actions of step (b) with call 2 and select VC2 and again update the timeline. Step (d) finishes the scheduling by scheduling call 3 on VC2 and finalizes the timeline for this example.

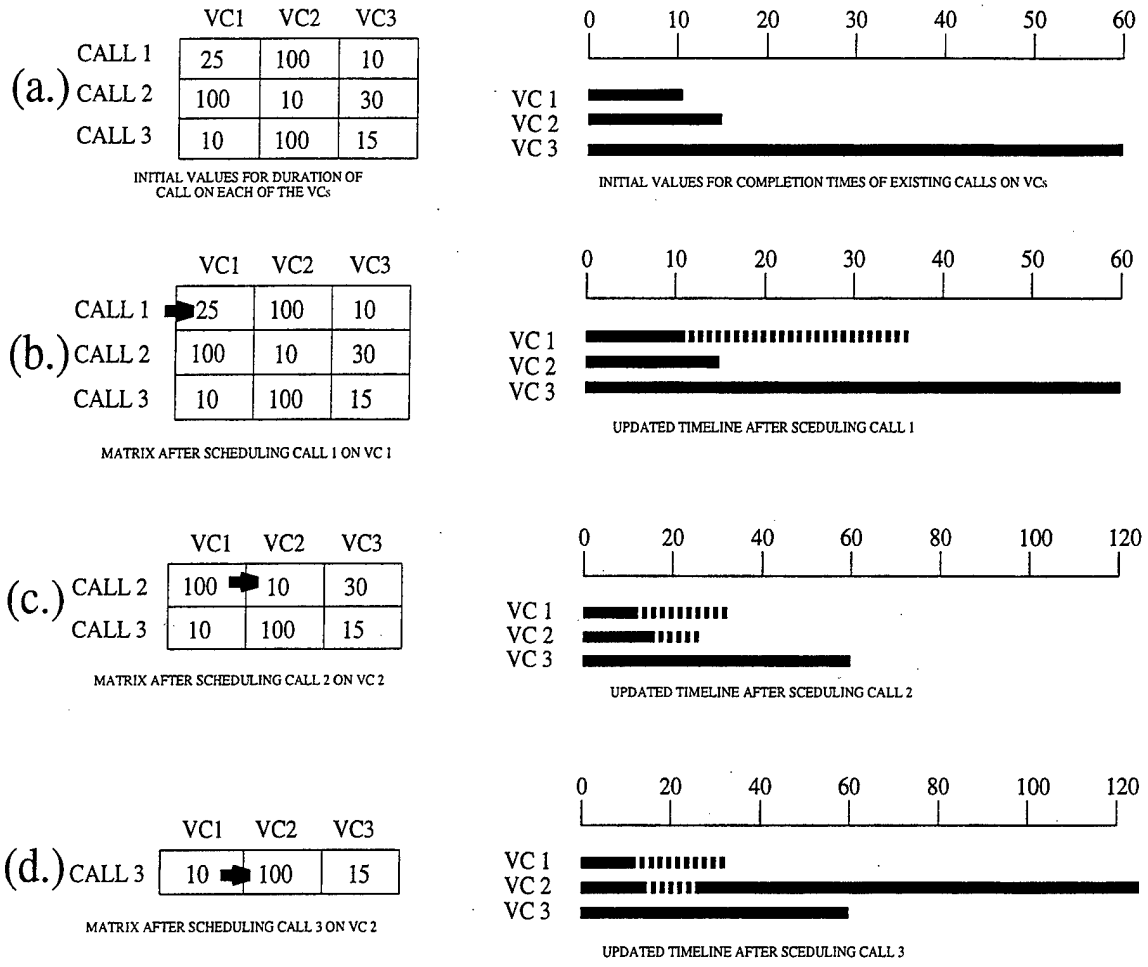


Figure 15. FIFO Algorithm Example

c. Strengths/Weaknesses

The strength of the FIFO algorithm is that it is very easy to implement. Its computational expense is $O(n)$, where n is the number of calls requiring service. The weakness of FIFO is that you may place calls on channels that are ill suited to carry them. The scheduling of call 3 on VC2 is an example of this problem. This selection would require 100 time units to complete. Selecting either of the other virtual

channels would cause an earlier overall completion time for all of the calls.

2. Best Fit

The **Best Fit** Algorithm is also a simple algorithm. Unlike the FIFO algorithm, this algorithm does not ignore the criteria that the user wants to optimize. In our scheduling problem, we attempt to minimize the time of which the last call completes. In the following example, the calls are scheduled in the order in which they are generated. For each call, the channel that would yield the minimum duration time, neglecting calls already scheduled or in progress, is scheduled to carry that call.

a. Best Fit Algorithm Specification

The following is a general algorithm for implementing the Best Fit Algorithm.

```
1.  Place all calls to be scheduled in the time-ordered queue,
Call_Queue, and set channel_avail_time[channel_number] = 0 for all
channels

2. While (Call_Queue not empty)

   3.  Remove first call in the queue.

   4.  Ignoring calls already in progress and those already
       scheduled, compute fastest time in which a channel can send
       this call.  Store index of that channel in schedule_index.

   5.  Schedule call that was removed from Call_Queue for
       channel[schedule_index].

   6.  Update channel_avail_time[schedule_index] by adding to
       it the amount of time needed to complete the removed
       call.

7. End While
```

b. Examples

Figure 16 displays an example of the steps for the execution of the Best Fit Algorithm. In step (a), each element of the matrix contains the required time

to send the call when scheduled on each of the virtual channels (VC). We depict the completion times for previously scheduled calls on each of the virtual channels using a timeline. In this example, virtual channels 1 through 3 will again finish at times 11, 15, and 60, respectively. In step (b), for call 1, we select the virtual channel with the smallest duration time. In this example it is VC3. Next we update the timeline and remove call 1 from the call scheduling matrix. In step (c), we repeat the actions of step (b) with call 2, select VC2, and update the timeline. Step (d) finishes the scheduling by choosing to place call 3 on VC3 and finalizing the timeline for this example.

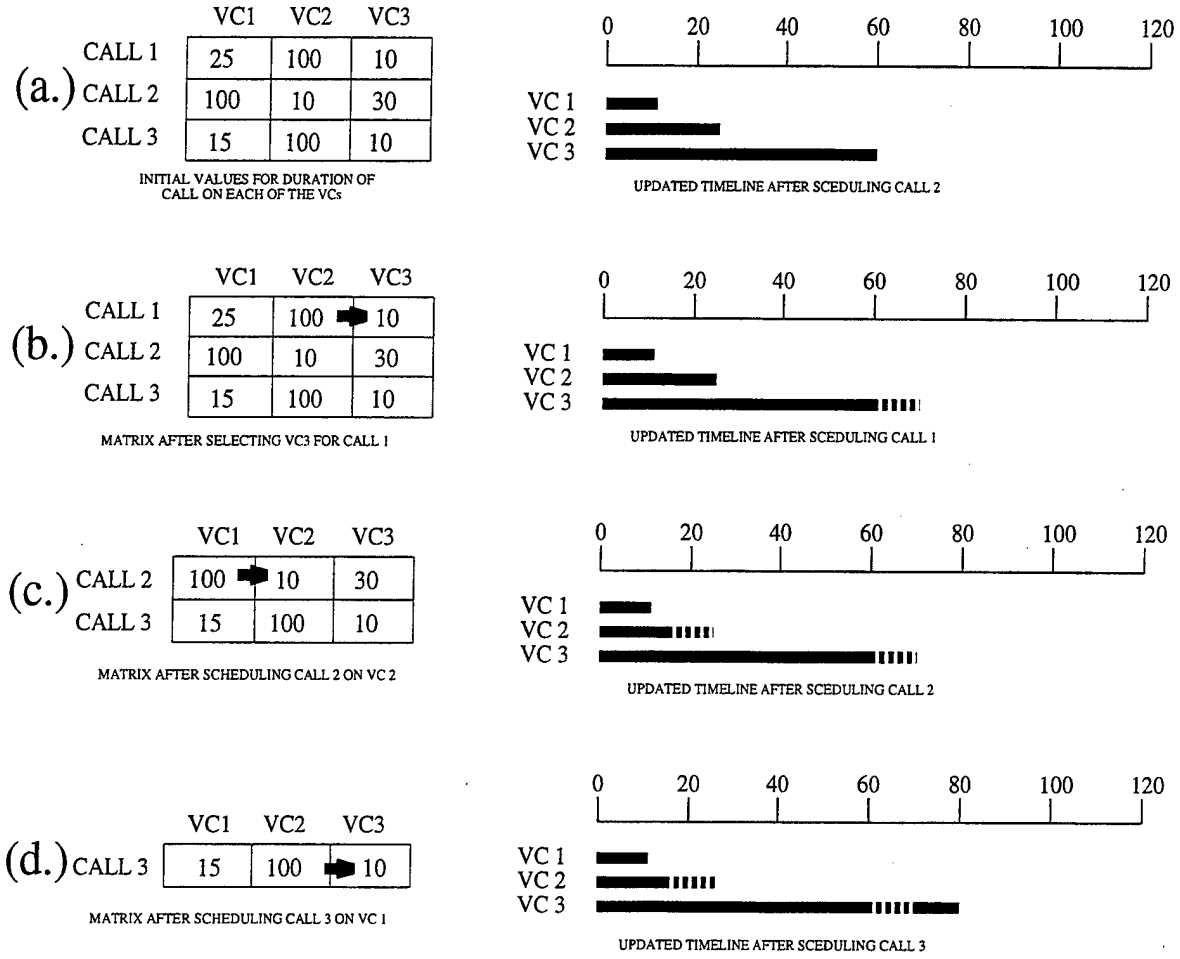


Figure 16. Best Fit Algorithm Example

c. *Strengths/Weaknesses*

The strength of this scheduling algorithm is that it considers the scheduling criteria. The algorithm is still relatively simple, requiring only a little computational overhead, however, it is more complex than FIFO, being $O(nm)$ where n is the number of calls and m is the number of channels. The major weakness is that the algorithm does not consider the load that has already been placed on each channel. We see this condition in the example with the scheduling of call 3 onto VC3 when the better choice would have been to schedule it on VC1.

3. $O(nm)$ Greedy Algorithm

The $O(nm)$ algorithm is a simple, yet very powerful means of scheduling calls onto virtual channels. The letter n again represents the number of calls and the letter m represents the number of virtual channels. This algorithm considers both the innate ability of a channel to support a transmission as well as the current load on each channel.

In the $O(nm)$ Greedy example that follows, the scheduling criteria is again defined as minimizing the time at which the last virtual channel completes. This algorithm is different from the previous scheduling algorithms because it bases its decision partly on completion time of each virtual channel.

a. $O(nm)$ *Greedy Algorithm Specification*

The following is a general algorithm for implementing the $O(nm)$ greedy algorithm for scheduling calls onto static virtual channels.

```
1.  Place all calls to be scheduled in the time-ordered queue,
    Call_Queue, and set channel_avail_time[channel_number] = 0
    for all channels.

2. While (Call_Queue not empty)

   3.  Remove first call in the queue.

   4.  Compute time at which this call would complete if assigned
       to each channel.  Determine the minimum of these completion
```

```
        times.  Store the index of the channel that yields the
        earliest completion time in schedule_index.

    5.  Schedule the call that was removed from Call_Queue for
        channel[schedule_index].

    6.  Update channel_avail_time[schedule_index].

7. End While
```

b. Examples

Figure 17 displays an example of the steps for the execution of the $O(nm)$ Greedy Algorithm. In step (a), the elements of the matrix contain the required time to send a call when scheduled on each of the virtual channels (VC). We depict the completion times for previously scheduled calls on each of the virtual channels using a timeline. In this example, virtual channels 1 through 3 will finish at times 11, 15, and 60, respectively. In step (b), we add the runtime from the matrix to the VC finish time on the timeline for the first call only. We also add these values to the first row because the $O(nm)$ Greedy Algorithm schedules the call with the earliest completion time, then moves on to schedule later calls. We also find that VC1 is the best choice because it has the earliest completion time, $t = 36$, so we update the timeline to reflect this choice. In step (c), we repeat the actions of step (b) with call 2 and select VC2 and update the timeline. Step (d) finishes the scheduling by selecting call 3 to be sent on VC1 and finalizes the timeline for this example.

c. Strengths/Weaknesses

The strength of the algorithm is that it requires relatively little overhead and executes rapidly. This greedy algorithm is computationally the same as the Best Fit Algorithm but is more computationally expensive than the FIFO example, $O(nm)$ vs. $O(n)$ respectively. This algorithm looks at the ramifications of scheduling each call on all of the available channels. It is a good algorithm for the user looking for modest levels of improvement with little processing expense.

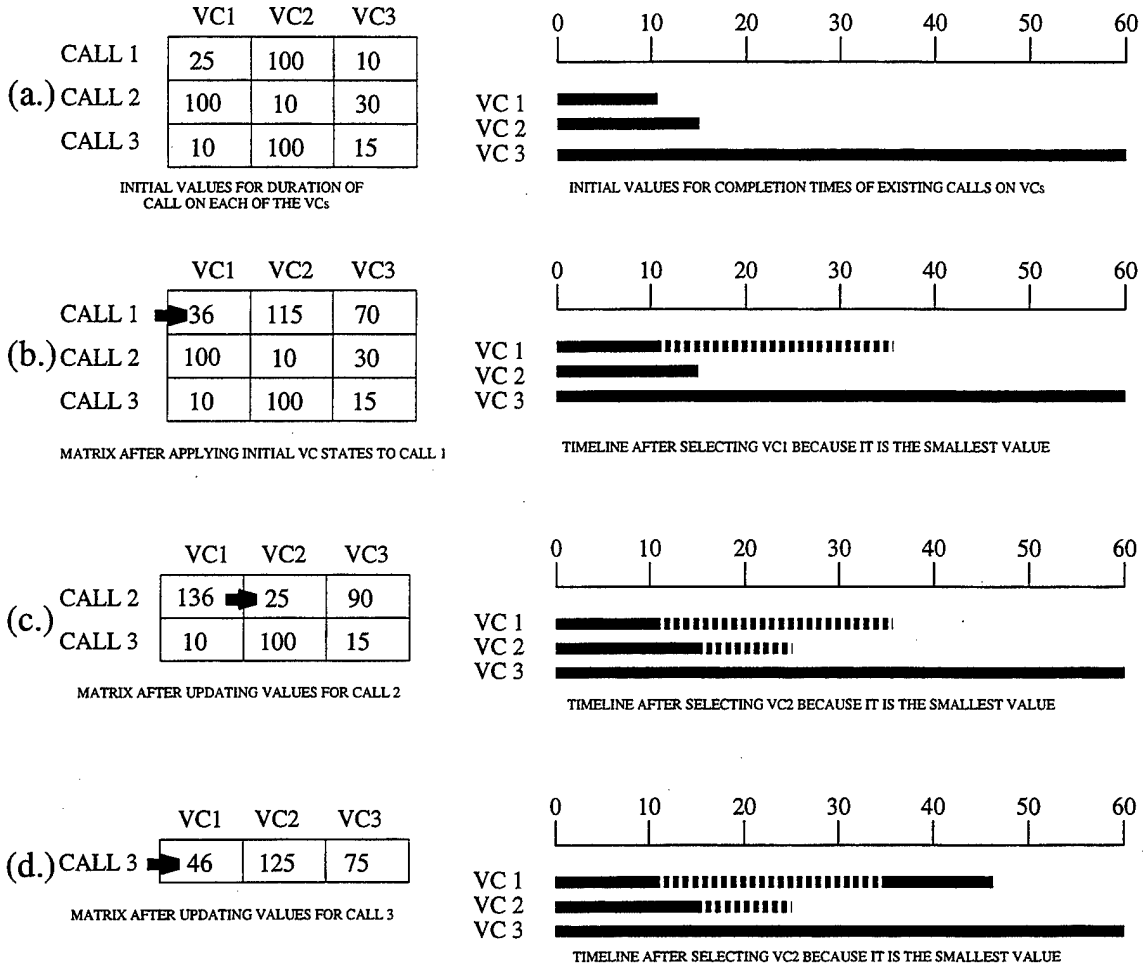


Figure 17. $O(nm)$ Greedy Algorithm Example

4. An $O(n^2m)$ Greedy Algorithm

This Greedy algorithm is more complex than the $O(nm)$ greedy algorithm. It requires an evaluation of predicted completion times for all calls in the scheduling matrix, and it generally produces better schedules at the expense of more overhead computation.

a. $O(n^2m)$ Greedy Algorithm Specification

The following is a general algorithm for implementing the $O(n^2m)$ greedy algorithm for scheduling virtual channels.

```
1.  Place all calls to be scheduled in the time-ordered queue,
```

```
   Call_Queue, and set channel_avail_time[channel_number] = 0
   for all channels.

2. While (Call_Queue not empty)

   3. Compute time at which each call would complete if scheduled
      immediately on each channel, including the time of any calls
      previously scheduled on the channel.

   4. Determine the (call, channel) pair with the earliest
      completion time.  Store the index of that channel in
      schedule_index, and the index of the call in call_index.

   5. Remove selected call from Call_Queue[call_index].

   6. Schedule call that was removed from Call_Queue for
      channel[schedule_index].

   7. Update channel_avail_time[schedule_index].

8. End While
```

b. Examples

Figure 18 displays an example of the steps for the execution of the $O(n^2m)$ Greedy Algorithm. In step (a), the elements of the matrix contains the required time to complete each call if it is scheduled on each of the virtual channels (VC). We depict the completion times for previously scheduled calls on each of the virtual channels in a timeline. In this example, virtual channels 1 through 3 will finish at times 11, 15, and 60, respectively. In step (b), we add the runtime from the matrix to the VC finish time on the timeline for all of the calls. This is in contrast to the $O(nm)$ Greedy Algorithm where we only applied values to the top row of the matrix. Next we iterate through each column of each row in the matrix to find the smallest value and flag it with an arrow. Then we iterate through all these flagged values and find the smallest of these values and flag it with a second arrow. We find that VC1 for call 3 is the best choice because it has the earliest completion, $t = 21$, so we update the timeline to reflect this choice. Finally, we delete call 3 from the matrix. In step

(c), we repeat the actions of step (b) by scheduling call 2 on VC2 and updating the timeline. Step (d) completes the example by scheduling call 1 on VC1 and finalizes the timeline.

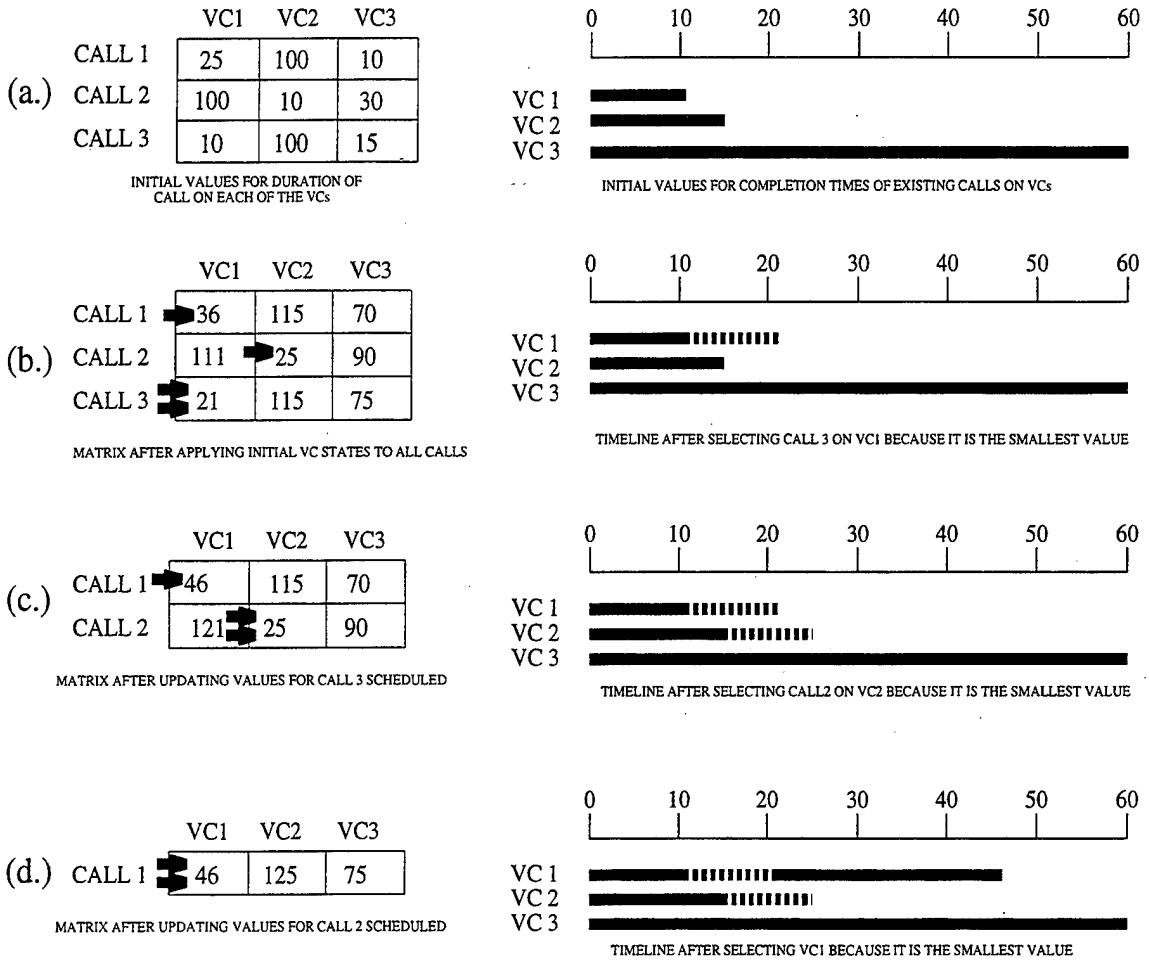


Figure 18. $O(n^2m)$ Greedy Algorithm Example

c. *Strengths/Weaknesses*

The strength of $O(n^2m)$ Greedy Algorithms resides in the more exhaustive search through the entire matrix to find the minimum value. This schedule is generally better than the $O(nm)$ Greedy Algorithm. The disadvantage of this greedy algorithm is that it requires much more computation to determine the scheduling. This computation requirement can cause delays which the user finds unacceptable.

D. SUMMARY

In this chapter we examined some of the algorithms available in the SmartNet Scheduler and provided examples to enhance the understanding of how they work. We also showed that they could be applied to scheduling calls on virtual channels, whereas SmartNet applies them only to scheduling jobs on high performance machines. We will use two of the algorithms in our simulations for this thesis. The are the FIFO and the $O(n^2m)$ greedy algorithms. The FIFO algorithm is the default used by OPNET and we selected the $O(n^2m)$ algorithm because it requires polynomial amount of computational overhead and considers both channel loads and characteristics. The comparisons of these two extremes will allow us to determine whether intelligent scheduling is worth the computational expense. In the next chapter, we describe our implementation of the BADD network using OPNET.

V. OPNET

A. CHAPTER OVERVIEW

This chapter describes **OPtimized Network Engineering Tools (OPNET)** from MIL 3 in enough detail to allow the reader to understand how we use this state-of-the-art network modeling tool to model the BADD network. The reader who is already familiar with OPNET can safely skip this chapter. Before describing OPNET's capabilities, we provide a quick overview of discrete event simulation. Readers familiar with this topic may proceed to the following section.

B. DISCRETE EVENT SIMULATION

Why are we using OPNET? What is Discrete Event Simulation? These are two excellent questions. The use of models in simulations provides a means of gaining understanding of complex problems. To construct the models requires the modeler to make simplifying assumptions in order to reduce the complexity and size of the object they wish to model while preserving the most important characteristics of the model. Models of physical objects, such as bridges, have been used to gain greater knowledge of the stresses associated with particular structures and materials. Some of the same modeling principles used with bridges can be applied to a communication network to gain an understanding of that network's dynamic behavior. The term dynamic means that the state of the network changes over time. As an example, changes occur to a network state when a new communication is placed on it or when an existing communication completes. One way to capture the dynamic nature of the network is to use a modeling tool, such as OPNET, that supports **Discrete Event Simulation.**

To understand discrete event simulation, the concept of a state must first be defined. A **state** is a collection of variables necessary to describe a system at a particular point in time. The "number of calls" in a wait queue is an example of a state variable for a communication network. In discrete event simulations, the state

variable's values change instantaneously at distinct points in time. Discrete event simulations are useful for characterizing networks because the number of calls in a system changes at discrete times when a call arrives or when a call is serviced by the network. The user is only interested in these specific points in time which are called **events**(because the state of network does not change continuously). Discrete event simulations use an **event list** to chronologically record when generated events are to take place. The time when state changes occur are maintained with a **simulation clock**. By advancing the simulation clock only when all scheduled events for a time take place, discrete event simulations can support the execution of concurrent events in the simulation. The ability to execute multiple events at the same time is a highly desirable feature because it allows the user to design more realistic simulations.

Figure 19 depicts the flow of control through the execution of a discrete event simulation. At time t = 0, the main program calls the initialization routine that sets the simulation clock, initializes system states, initializes the empty event list, generates the first event, and places it in the list. After the initialization routine is complete, the main program invokes a **Simulation Kernel** that takes the first event off the time-ordered list, advances the clock to the time of this removed event, and invokes the **Event Handler** corresponding to this event. As an event handler executes, it generates new events and places those in the event queue. Also, during the execution of an event handler, the system state is updated and statistical counters are maintained. When an event handler completes, control is returned to the simulation kernel. A program implements simulation termination in any of three ways: (i) when the simulated time reaches a certain value, (ii) when a statistical counter reaches a particular value, or (iii) in unusual circumstances, when the event list is empty. When a simulation completes, it computes the statistics of interest and writes a report for analysis. We presented a very high level view of discrete event simulations and the interested reader can consult textbooks for a more detailed discussion of this topic. [Ref. 17]

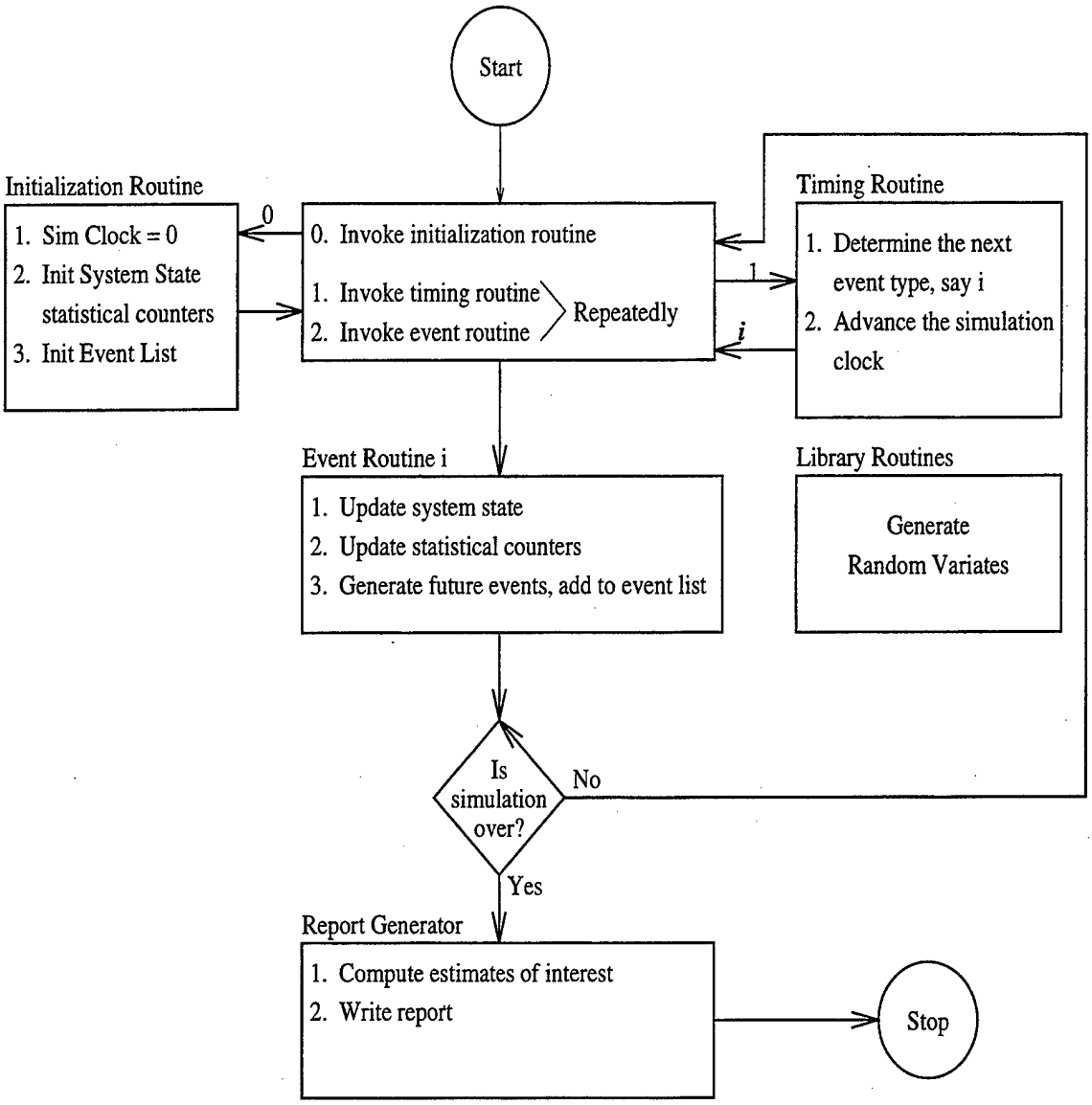


Figure 19. Discrete Event Simulation Flow Control (From [Ref. 17]).

C. OPNET OVERVIEW

OPNET is a robust and comprehensive engineering system capable of simulating large communication networks with detailed protocol modeling and performance analysis. OPNET is an object-oriented modeling system that allows the user to easily traverse through a multi-level network model to perform refinements and modifications in support of an iterative design approach [Ref. 4]. Figure 20 shows OPNET's Hierarchy where the **Network Layer** is built from **Sub-Networks**, Sub-Networks are built from **Nodes**, and the Nodes are built from **Process Models**.

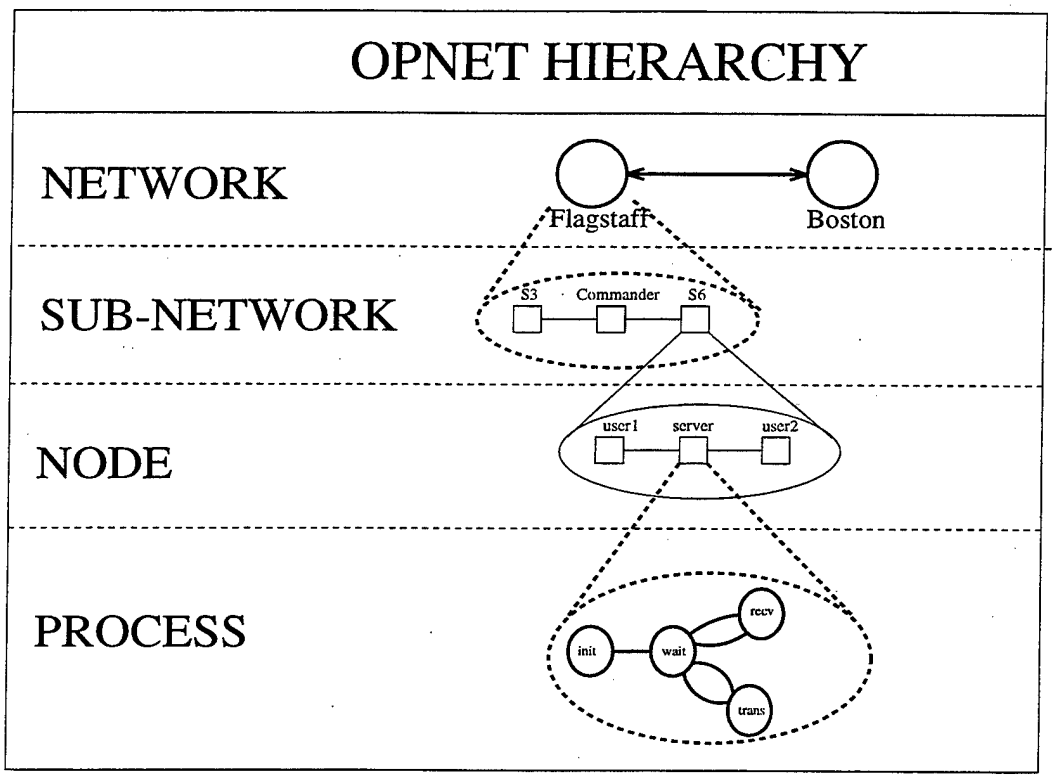


Figure 20. OPNET Hierarchy (Derived From [Ref. 4]).

The Network Layer through the Node Layer focus on the topography of the network. This allows the user to visualize the structure and interactions between objects in the network at finer levels of abstraction as we move down through the layers. While the Network Layer in our example only shows a geographic relationship of the network between Flagstaff and Boston, the Node module provides much greater

detail by indicating the functionality of the node by using a naming convention (i.e., the server node with the name S6).

The Process Model defines the behavior for processors and queue modules in OPNET. A process is an instance of a process model and operates within an OPNET processor module. In Figure 20, we show that the server processor module has various states within its process model (init, wait, receive, and transmit). These states, and the transitions between them, reflect the specific behaviors that occur repeatedly in a server. Now that the reader has a general understanding of the hierarchical framework, we describe how OPNET supports discrete event simulation.

D. OPNET SUPPORT OF DISCRETE EVENT SIMULATION

OPNET supports discrete event simulation of network loads, allowing the user to easily modify attributes and providing analysis tools with which to examine network performance, either globally or at at specific points in a network. OPNET uses a **Simulation Kernel** to run the simulation by controlling the addition and deletion of events to a single event list. The simulation kernel and process models can generate events for addition to the event list using **OPNET Kernel Procedures (KPs)**. KPs also allow the user to gain information about the event list, for example, the time of the next scheduled event. Events can be added to the event list for the current simulation time or for any time in the future. (Events from the past cannot be added to the list.) Figure 21 illustrates an OPNET event list. The boxes in the figure represent events scheduled on the list. We have highlighted OPNET's ability to support multiple events on the list which are to occur at the same time. It accomplishes concurrent execution by executing all events scheduled for the same time before advancing the simulation clock to the time of the next event on the list. We also highlight the fact that the time between events on the list can be variable which is characteristic of using Next-Event time advance with simulation time.

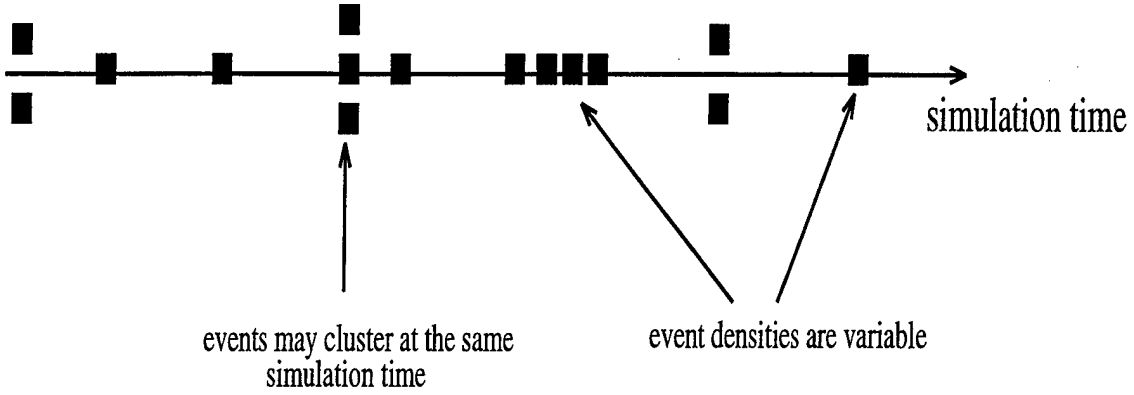


Figure 21. Distribution of Events on an Event List (From [Ref. 4]).

The user accesses the KPs within processes using the Process Editor. The Process Editor will be discussed in greater detail later in this chapter; however, we will discuss its interaction with the simulation kernel now. As we stated earlier, the simulation kernel adds events to the event list. This is accomplished when the kernel receives interrupts. Interrupts can come from the kernel itself or from the process models. Examples of kernel interrupts are those that signal the beginning and ending of the simulation. Examples of process models calling interrupts include one that causes a process to transition from one state to another after a specified delay, one that allows a process to signal another to perform an operation, and one that sends a packet to another process and notifies that process when the packet arrives. For example, the last interrupt described above would be called in the transmit state found in Figure 20. In the following section we will discuss the Process Editor in more detail, describing how to use it to construct finite state machines. We now examine the tools OPNET provides to build network models and simulate network loading.

E. OPNET TOOLS

OPNET supports network design, testing, and analysis with the eight tools listed in Table VII. All of OPNET's tools are available through an advanced graphical user interface (GUI) known as the **MIL 3 User Interface**. It provides support

utilities in each of the tools that are accessed via **action buttons**. Some of these utilities are common to all of the editors, such as the utilities that read and write files and print reports and graphics. Other utilities are unique to a particular tool. We will only address the most commonly used utilities within each of the tools. Now we will describe the function of each of these tools.

Tool	Purpose
Network Editor	Design Overall Network Topology
Node Editor	Design and Edit Nodes in Network
Process Editor	Design and Modify State Diagrams of Nodes
Parameter Editor	Modify Parameters of Links and Packets
Probe Editor	Set Probes to Collect Network Statistics
Simulation Tool	Configure and Run Simulations on Network
Analysis Tool	Evaluate Data Collected during Simulation
Filter Editor	Define numerical processing filters to take multiple inputs and produce an output

Table VII. OPNET Tools (Derived From [Ref. 4]).

1. The Network Editor

The Network Editor is used to construct the user's network models. Its main purpose is to define, at a high level, the topology of the network. Networks in OPNET consist of communication nodes connected by point-to-point links, busses, and radio links. The nodes and links can be grouped together to form subnetworks. These subnetworks can then be viewed as single objects within a larger network, rendering simpler models that are more easily understood.

The most commonly used utility in the Network Editor is an **Object Palette**. It allows the user to configure a workspace to contain only those objects that are necessary for the network currently under development. For example, when designing an ATM network, there is no need for models supporting Ethernet. This utility also includes a **Rapid Configuration** constructor to allow the user to quickly build the most common networks such as a star or bus topology. Also useful is the **Carto-**

graphic Background that allows the user to load maps as a background for their networks.

Figure 22 displays the Network Editor, with the Object Palette open, and an example network model.

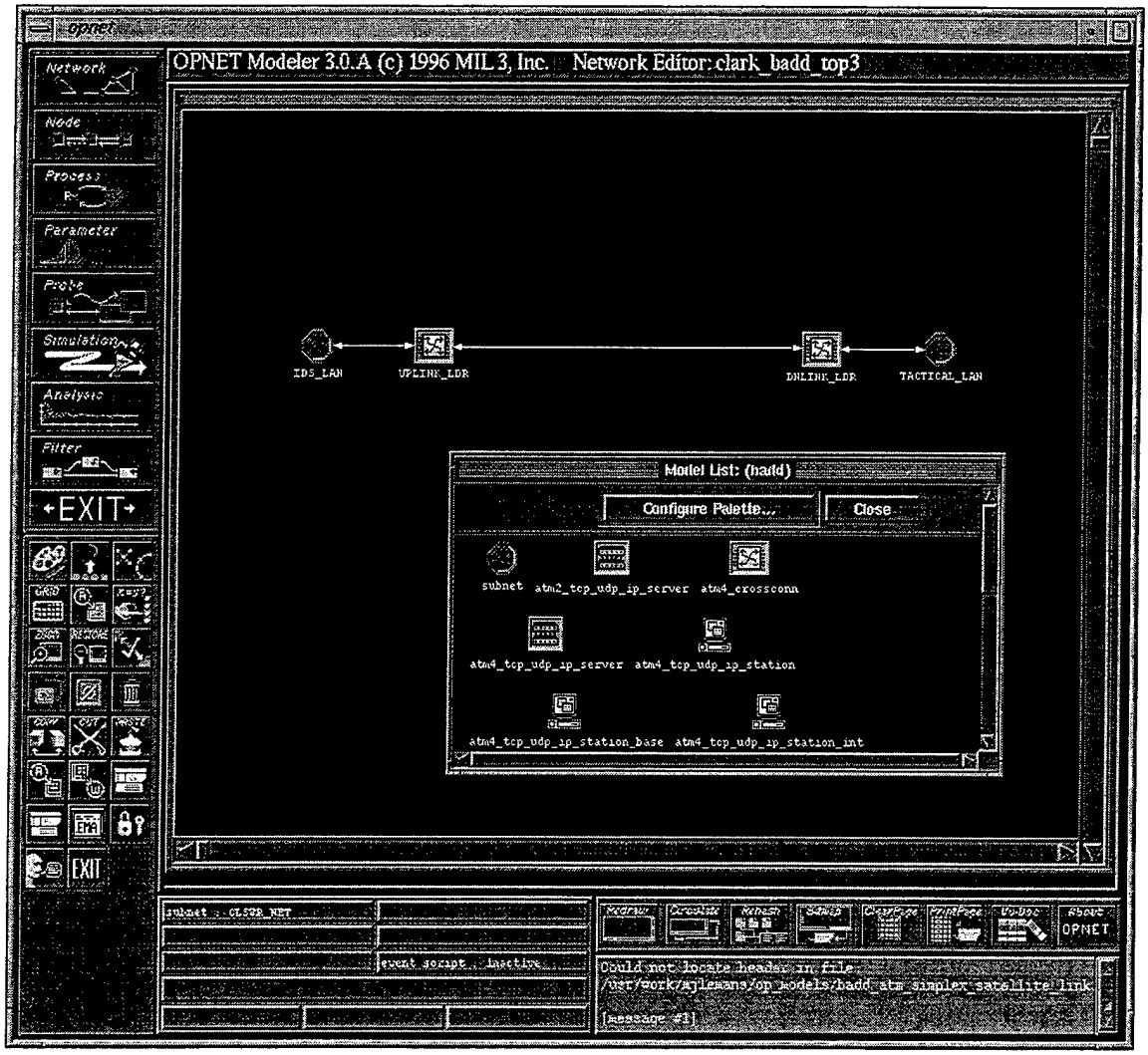


Figure 22. OPNET Network Editor. From [Ref. 7]

2. Node Editor

The Node Editor is used to construct models of nodes within the network. It provides magnification to the network model, showing more details of the network

topology at a local level. Nodes are created by connecting **modules** (i.e., processors, queues, data generators, receivers and transmitters) together with **packet streams** and **statistic wires**. Packet streams allow packets to be sent between the output interface of a source module and the input interface of the destination module. Statistic wires support the flow of numerical data by connecting output statistics of a source module to the input statistic of the destination module. The most commonly used utilities are listed in Table VIII.

Action Button	Purpose
Processor	General purpose, programmable object, whose behavior is specified by Process Modules
Ideal Generator	Used to generate random packets
Clock Generator	Used to generate packets aligned with a synchronous clock
Queue	Used to queue up packets awaiting service
Pt-Pt/Bus Receiver	Used to receive packets
Pt-Pt/Bus Transmitter	Used to transmit packets

Table VIII. OPNET Node Editor Action Buttons (Derived From [Ref. 4]).

The Node Editor allows the user to create and assign specific naming attributes to the modules that comprise their network. An example would be assigning the name "server" to a user defined module. The construction of a node provides a skeleton of the modules required to accomplish its particular function (i.e. processor, generator, or queue). To fill in the skeleton we use the finite state machines found in the Process Editor.

3. Process Editor

The Process Editor is the most important component of OPNET. The Process Editor allows the user to create state diagrams for the processor and queue modules defined in their Nodes. State diagrams are useful when modeling networks because of the nature of networks. Networks are dynamic, and transition between a finite

number of well-defined states (i.e. idle, transmitting data, receiving data). Networks do not create new states over time. They rotate between a finite number of states based on responses to specific stimuli. State diagrams allow the user to abstract these states, and the transitions that cause them to move between states, into a form that is easily understood. In addition, state diagrams are easily integrated into discrete event simulations because the states of a process model can easily be correlated to events in a simulation. A more detailed explanation of state diagrams is found in Hopcroft and Ullman's book [Ref. 18].

Process Models represent the logic embodied in the communications, hardware, network protocols, and algorithms that comprise the network. It is important to note that Process Models define the states of processors and that these two terms are not interchangeable. To use an analogy, the Nodes in a network are similar to the skeletal structure of a human. The process models of a network provide movement in the network states just as the muscles and tendons that attach to the bones on the skeleton make movement possible in a human. A processor is similar to the shell of an non-descript muscle. It requires further behavioral definition, through the use of state diagrams, to describe its specific function.

The Process Editor uses **Proto-C**, a language that combines graphical state transition diagrams, embedded C language data items and statements, and a library of kernel procedures to provide commonly needed functionality for modeling communication networks and information processing systems. The process editor allows the user to view this code using the action buttons listed in Table IX.

Figure 23 displays the Process Editor with a state diagram as a traffic generation module. OPNET state transition diagrams are composed of both state and transition components. In the next portion of this section we will describe the attributes of these components that allow the user to define protocols in support of discrete event simulations.

States may have associated with them actions to be executed upon entry into

Action Button	Purpose
Edit State Variables	Allows declaration and modification of state variables that represent persistent counters, routing tables, etc. State variables can only be modified by explicit assignment statements by the process itself.
Edit Temp Variables	Allows declaration and modification of temporary variables used for "scratch" storage in calculations (non-persistent).
Edit Header Block	Allows declaration and modification of global variables.. Global variables allow multiple processes to access them creating dependencies among processes
Edit Function Block	Allows editing and declaration of functions that support recurring calculations
Edit Diagnostic Block	Used to indicate which state variables should be monitored. Used for debugging.

Table IX. OPNET Process Editor Action Buttons (Derived From [Ref. 4]).

the state, **(called enter executives)**, and when exiting the state, **(called exit executives)**. Calls to OPNET defined functions, or C-code written by the user, cause the execution of these actions.

OPNET classifies states in a process model to be either **forced** or **unforced**. Unforced states allow a pause in execution between the enter executives and the exit executives. In other words, once the enter executives of an unforced state are completed, the simulation kernel can take control and execute another event. While another event is being processed for another process, this process remains in the unforced state. Eventually the simulation kernel will regain control again and signal the process to execute its corresponding exit executives. Forced states do not permit the simulation kernel to execute between the enter and exit executives. Since forced states do not relinquish control to the simulation kernel, OPNET requires every process to have at least one unforced state.

Transition for OPNET are classified either as **conditional** or **non-conditional**. Conditional transitions occur when a condition is placed on a transition, such as the

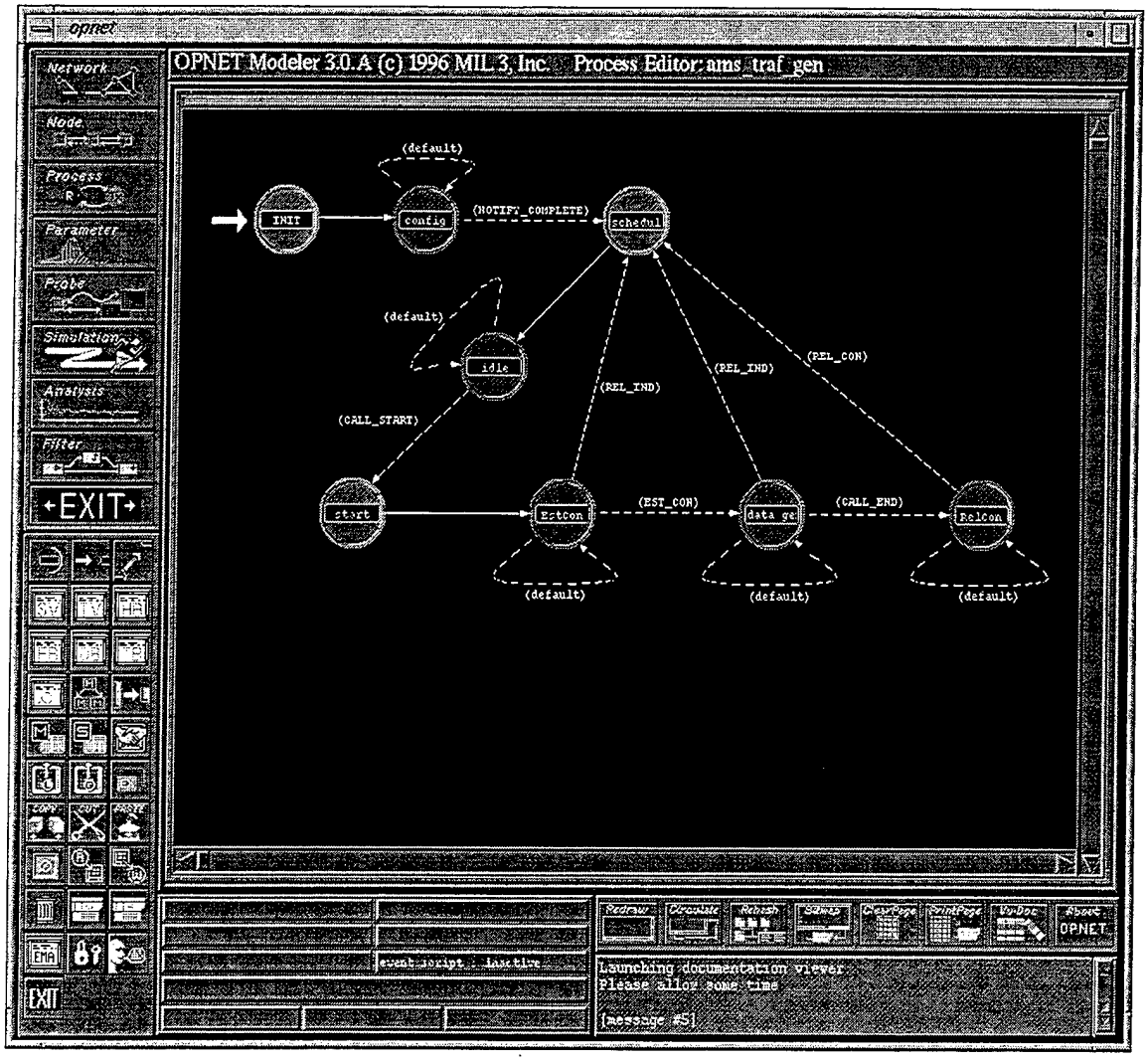


Figure 23. OPNET Process Editor. From [Ref. 4]

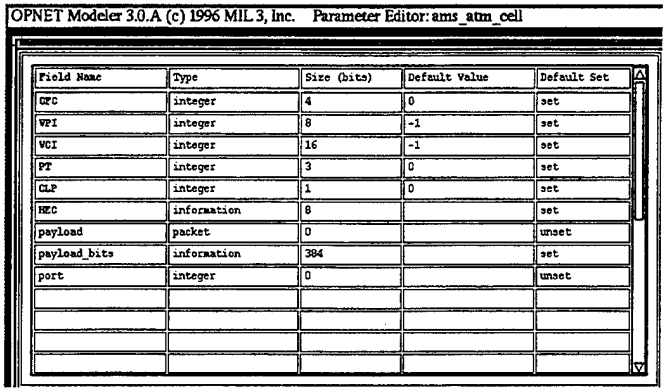
CALL-START condition between the **IDLE** and **START** states in Figure 23. In this example, a call-start signal must be received for the transition to occur. The transition between the **INIT** and the **CONFIG** states in the same figure depicts a non-conditional transition. OPNET uses dashed arcs to depict conditional transitions and solid arcs to depict non-conditional transitions.

In order to support concurrent operations in a simulation, OPNET allows processes to spawn new processes, called **dynamic processes**. There is no constraint on the number of dynamic process spawned in support of a simulation. In the next

chapter we will discuss in detail the use of dynamic processes to support our model of the BADD Network.

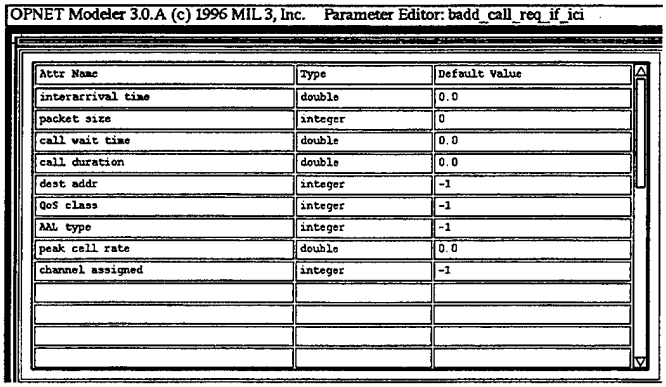
4. Parameter Editor

The OPNET Parameter Editor provides easy access to several other tools that can be used to edit complex objects. Some examples of these complex objects include the Packet Formats shown in Figure 24, the Interface Control Information Format (ICI) shown in Figure 25, and the Link Model shown in Figure 26.



Field Name	Type	Size (bits)	Default Value	Default Set
GFC	integer	4	0	set
VPI	integer	8	-1	set
VCI	integer	16	-1	set
PT	integer	3	0	set
CLP	integer	1	0	set
HEC	information	8		set
payload	packet	0		unset
payload_bits	information	384		set
port	integer	0		unset

Figure 24. OPNET Packet Format (From [Ref. 4]).



Attr Name	Type	Default Value
interarrival time	double	0.0
packet size	integer	0
call wait time	double	0.0
call duration	double	0.0
dest addr	integer	-1
QoS class	integer	-1
AAL type	integer	-1
peak cell rate	double	0.0
channel assigned	integer	-1

Figure 25. OPNET Interface Control Information Format (From [Ref. 4]).

5. Probe Editor

The Probe Editor is used to specify locations where the user would like to collect data. Probes are specified prior to running a simulation. Several types of

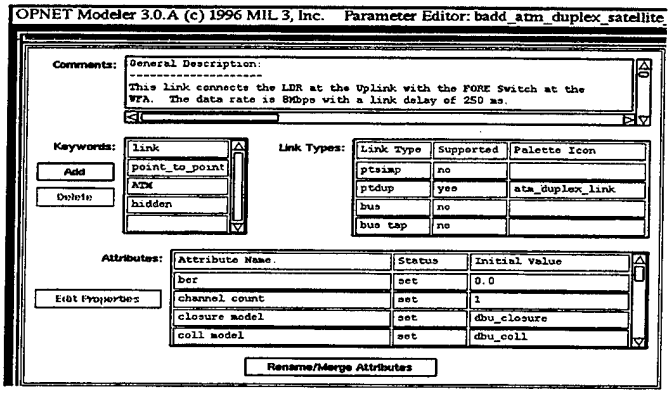


Figure 26. OPNET Link Format (From [Ref. 4]).

probes and their functions are listed in Table X.

Action Button	Purpose
Node Statistic	Supports collection of built-in and user-defined statistics
Link Statistic	Supports collection of built-in statistics on link objects at the network level
Global Statistic	Supports collection of values from multiple objects throughout the network
Simulation Attribute	Records scalar statistics for post-simulation analysis of simulation outputs
Automatic Animation	Provides programming-free animation of packet flows at network and node level; node movement at network level; and state transitions of a process
Custom Animation	Provides custom animation within process models and link models (requires user programming)
Statistic Animation	Provides programming-free animation depicting statistics through the simulation

Table X. OPNET Probe Editor Action Buttons (Derived From [Ref. 4]).

6. Simulation Tool

OPNET provides a user-friendly graphical interface with which the user can run multiple simulations simultaneously. OPNET also allows the user to run simu-

lations from the command line. Running simulations from the command line can be useful for saving simulation runs and traces into files for later analysis. The command line interface also provides a number of standard fields that support unattended execution of simulations. These fields are summarized in Table XI.

Field	Purpose
Network	Network Model used for simulation
Probe File	Name of Probe File specifying data collected during the simulation
Vector File	Name of file to store vector output generated by simulation
Scalar File	Name of file to store scalar output generated by simulation
Seed	Allows user to vary seed to gain statistical confidence in results
Duration	Maximum duration of simulation in simulated seconds
Application Specific	Allows user to define values for promoted attributes of network and process models

Table XI. OPNET Simulation Editor Fields (Derived From [Ref. 4]).

7. Analysis Tool

The Analysis Tool supports the display of data stored in two types of files, the **output vector** files and the **output scalar** files. Output vector files are collections of pairs of real values. Output vectors contain the information from the simulation and are used as input to construct **traces**. Traces are ordered sets of abscissa and ordinate pairs, called entries (traces are similar in structure to vectors [Ref. 4]). OPNET allows the user to view vector information in **panels**. A panel is a two-dimensional time-series graph format with the horizontal axis value representing an independent variable, usually simulation time, and the vertical axis representing a dependent variable. The analysis tool also provides the ability to plot multiple traces on a single panel. Such plots are useful for comparing multiple different runs. Additionally, this tool allows

the user to vary the scale and range of the axes to focus on specific segments of interest. Figure 27 shows an example of output from the Analysis Tool.

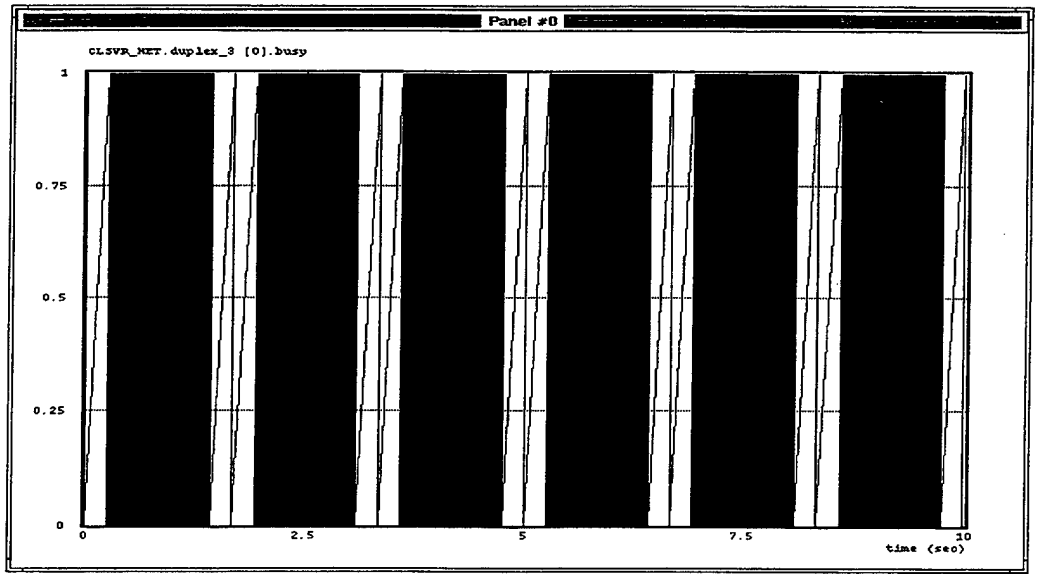


Figure 27. OPNET Analysis Tool (From [Ref. 4]).

Output scalar files contain values that are accrued over multiple simulations. Examples of values that can be found in these files are average wait times, peak sizes of queues, and standard deviations and means for end-to-end delays.

8. Filter Editor

The Filter Editor allows the user to edit models that define the mathematical processing of simulation results. A filter model can be thought of as a single system that defines both inputs and an algorithm for computing the output. Figure 28 depicts a schematic of an implementation of a general filter editor. OPNET supports a hierarchical structure of filtering where multiple computations are executed to derive a desired value. This desired value is derived by linking the outputs from multiple filters together using filter **connections**. The composition of multiple filters is called a **macro filter**. The editor also contains pre-defined filters that the users can use to build macro filters. These filters include an Adder, Gain, Multiplier, Mean, Average,

Sum, Differentiator, Integrator, Exponentiator, Logarithm, Delay and Time Window Filter.

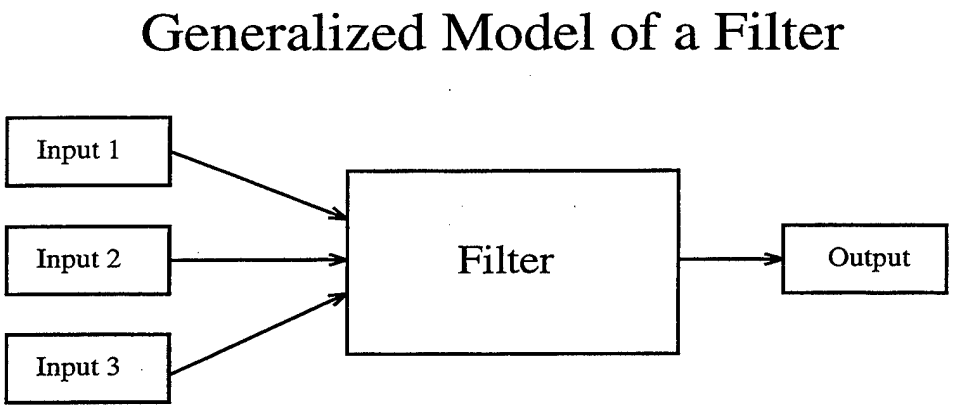


Figure 28. Generalized OPNET Filter From [Ref. 4]

F. SUMMARY

OPNET is a sophisticated network engineering tool as seen from the review of its tools above. These tools provide us with a framework for developing new variations of protocols and testing these variations with simulations. In the next chapter we explain how we used OPNET to model BADD and intelligent scheduling of the ATM virtual channels contained in the BADD network.

VI. DESIGN OF BADD NETWORK WITHIN OPNET

A. INTRODUCTION

The goal of this work is to design a model of the BADD network with OPNET in order to simulate and evaluate the performance of different network channel scheduling algorithms. This chapter describes, in detail, our OPNET design of the BADD network, in particular, the design of static channels within the BADD's ATM network, the design and integration of multiple call requesters, and the design of several different BADD call schedulers.

This chapter includes numerous diagrams of nodes, modules, and processes within OPNET. The state names in many of the diagrams are truncated by OPNET's graphic print utility. The complete state name is used in our descriptions and all OPNET reports.

Processes are not included in node diagrams because processes are dynamic. Because we do not know the exact timing or the exact number of calls generated during a given simulation, processes are created dynamically as needed. These dynamic processes allow concurrent processing of multiple call requests.

As stated in the Chapter VI, OPNET supports multiple events occurring in the simulation at the same time. OPNET uses dynamic processes to support this concurrency feature. The distinction between normal processes and dynamic processes can best be drawn from the following analogy. Regular processes reflect the hardware in a system, they can only accomplish one action at a time. Dynamic processes reflect actions that are running independently.

B. BADD NETWORK

Our first objective was to use OPNET to design and implement a model of the BADD network. Because the overall BADD network is extremely complex, we

scaled our design to focus on the ATM backbone between the IDS and the WFA. This portion of the network contains the GBS (satellite) ATM link. As detailed in chapter III, this link reduces the effective ATM network capacity to 8 Mbps. We will use our model to simulate various scheduling policies to determine which are best at directing the network traffic through this bottleneck.

The GBS link provides only simplex communication. However, OPNET currently only supports duplex ATM channels. The subsection titled **Network Communication Links** describes how we modified OPNET's duplex communications links to closely approximate the GBS's simplex links. A high level view of our initial BADD network model is given in Figure 29.

The initial BADD model was developed using OPNET's ATM modules interconnected to reflect the BADD architecture. OPNET provides various ATM modules to support the standard ATM network protocol. Because BADD utilizes ATM in such an unusual way, we selected OPNET's simplest ATM modules and modified them as necessary to support our model. A full description of each of the components and network modifications is given in the remainder of this section.

In the subsections below we will be describing the various modules that comprise the four nodes in Figure 29. Quite often a particular module may be used by more than one node. When we describe such a module, we will note which other nodes it is used in and provide a general description of its function and structure. For example, in describing the ATM_mgmt module in the IDS LAN traffic source node, a module that is also present in all of the other nodes, we will say that certain states may be used to decide whether the final destination of a message is that node or another.

1. IDS LAN Subnet

As described in chapter III, the IDS consolidates information to be broadcast over the GBS system. For simplicity, we first modeled the IDS LAN as a single traffic source node. This initial traffic source generates constant sized data packets at a constant rate; a decision we made to keep the model simple until the overall network

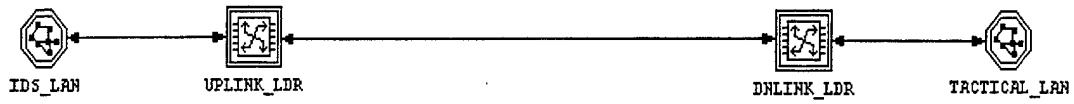


Figure 29. Designed BADD Network within OPNET.

model functioned properly.

A detailed view of the IDS LAN node is shown in Figure 30. This node consists of four basic components: a module to generate calls, an AAL module, several ATM modules and a transmitter-receiver pair.

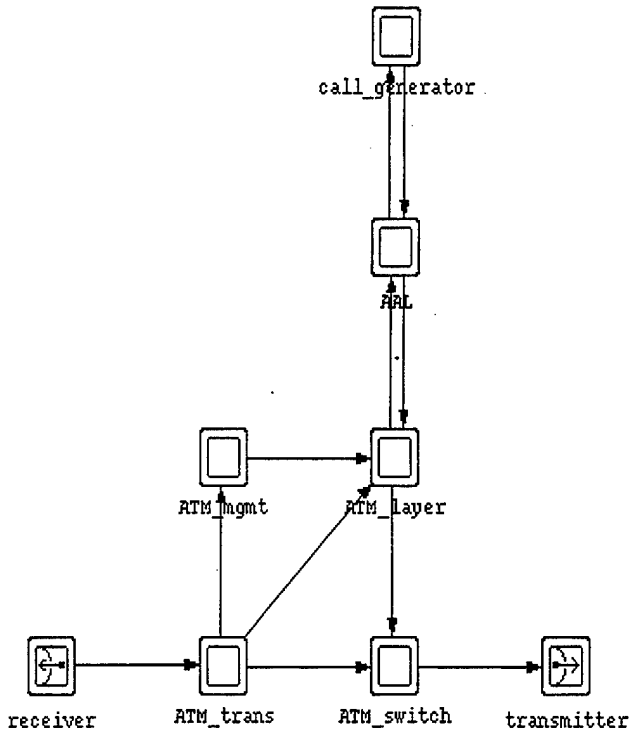


Figure 30. Initial BADD traffic source node with single call_generator.

This node communicates with other nodes in the network through the transmitter-receiver pair (transceiver). A transceiver is required for each communication link.

a. The Call Generation Module

The function of the call_generator module is to generate requests to send data as well as to generate the actual data packets to be broadcast over the simulated BADD network. The state diagram illustrating the action of this module is shown in Figure 31. A description of these states is given below.

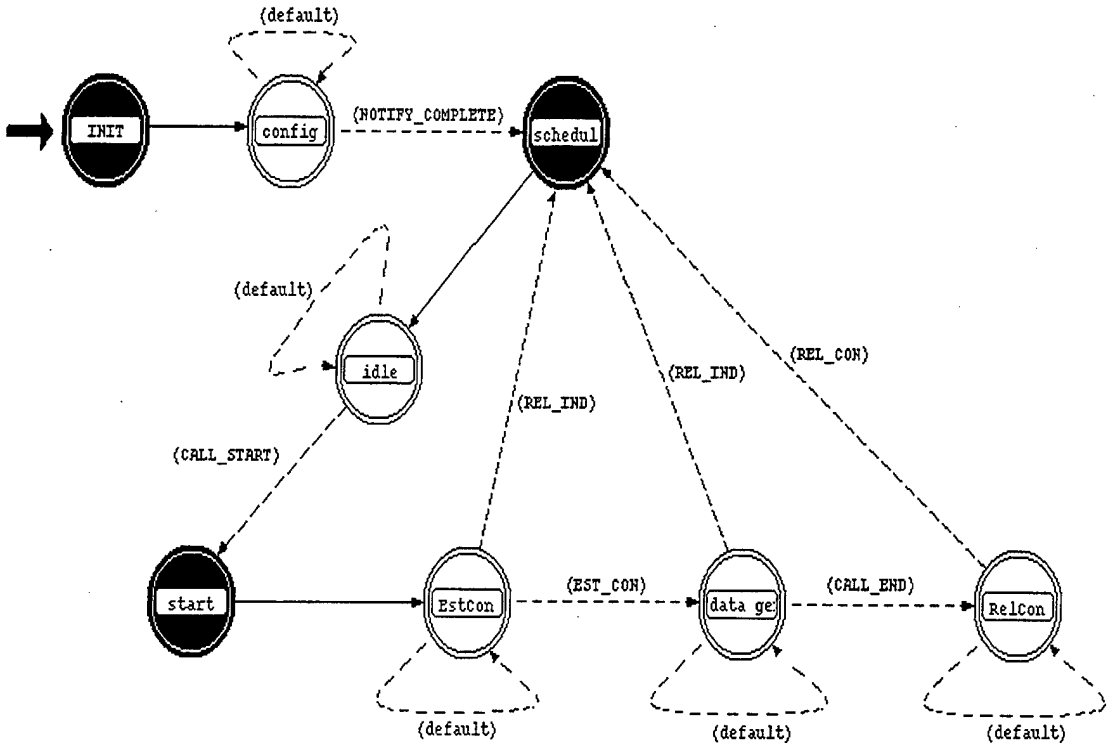


Figure 31. State diagram for initial BADD call_generator.

- **INIT**. When the program is in this state it initializes the variables local to this module. Using functions provided by OPNET, it also reads the model attributes for this module. The model attributes include the mean duration of the call, the mean call wait time, the mean interarrival rate and the mean packet size. These attributes define the type of calls generated by this module. The use of each of these variables is described in other states within this module.
- **config**. From the **INIT** state, the process transitions to the **config** state where it first broadcasts the IDS LAN's network management information, such as object ID, module ID, and module type, to a corresponding module inside all other nodes. Then it waits to receive their information. The waiting is shown in

our diagram by the transition labeled **default**. We note that while the process is in this state, it can be interrupted by the simulation kernel. As described in Chapter V, this is called an unforced state.

- **schedule**. When the module enters this state it generates an event, in the future, at which time the next request will be generated. The time of the generated event is controlled by the mean call wait time distribution read while the process was in the **INIT** state.

- **idle**. The process remains in the **idle** state, an unforced state, until the generated request time arrives. This wait time was determined in the **schedule** state.

- **start**. When the time arrives for the scheduled call to occur, this process transitions to the **start** state. The module initiates the call by creating a call descriptor **Interface Control Information** (Ici) packet and forwarding the Ici to the remote AAL module. The Ici is an OPNET defined, multi-member data item used to pass control information between different modules and processes. The Parameter Editor, as described in Chapter V, can be used to modify the Ici format. For this module, we used the predefined Ici, ams_if_ici packet. This Ici contains the call's destination address, the call's AAL type, the call's QOS class, and the call's traffic contract requirements.

- **EstCon**. The module waits in this state for the AAL module to respond to the call request. If the AAL module responds with a call release indication because the network cannot support the call request, this module drops the call request and transitions back to the **schedule** state. If the AAL module responds with a call connection signal, the module generates a delayed event that signifies the end of the call. Rather than count packets, this module generates packets until the end of the call event arrives. The delay time for this event is generated using the mean call duration described earlier. Finally, it generates an event corresponding to the generation of the first data packet which, after an appropriate wait time, causes a transition to the **data gen** state.

- **data gen**. While in this state the process generates data packets and forwards the data packets to the AAL module. The data packets are generated at the size and rate defined by the model attributes mean packet size and mean interarrival time, respectively. The module remains in this state until the end of the call event arrives or the AAL module signals a release of the data channel. If the call request is completed, this state sends a call release signal to the AAL module and transitions to the **RelCon** state. If the AAL module sent a call release signal, meaning the network dropped the channel, this state stops sending data

packets and drops the call request. The call release signal causes a transition back to the **schedule** state for scheduling the next call.

- **RelCon**. When in this state, the module waits for the AAL module to signal a Release Confirm acknowledgment, meaning the ATM network completed the transmission of all data packets and has released the channel resources. Upon receiving Release Confirm, this module transitions back to the **schedule** state.

b. AAL Module

The AAL module serves as the interface between the call_generator and the ATM modules. While in this section we focus our descriptions on the IDS LAN node, this module is also used in the TACTICAL LAN node. This module coordinates the signaling of call requests from the call_generator and it segments the call data packets into 48-byte segments for transport across the ATM network. The AAL process state diagram is shown in Figure 32. A description of these states is given below.

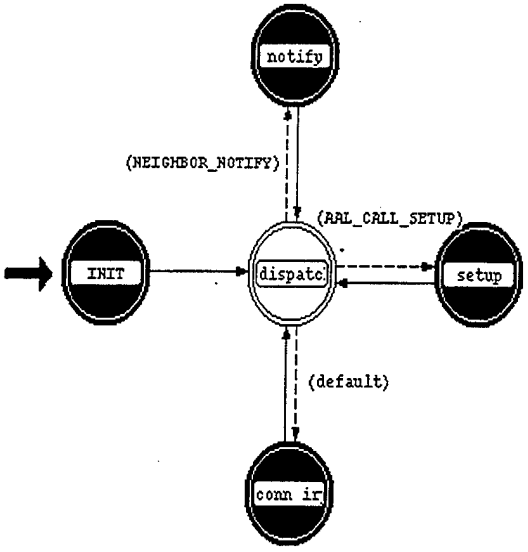


Figure 32. State diagram for AAL module.

- **INIT**. This state is used to initialize numerous variables within the process model. Additionally, while in this state, the module creates and initializes memory that is shared with all dynamic processes spawned by this module.

- **dispatch**. In this state, the module distributes appropriately all requests received by the AAL module. The distribution is controlled by the type of request received and the data within the request.

- **notify**. The process in this state broadcasts node descriptor information to all other nodes within the simulation network.

- **setup**. Upon receipt of a call request from the call_generator or the network, this state is used to spawn a dynamic **Signaling AAL** (SAAL) process to handle that request. It then passes the call descriptor Ici to the newly created process. Because the SAAL process is a dynamic process, it is not shown in Figure 31. (The SAAL process is described later in this section.)

- **conn irpt**. When in this state the module receives and processes signals pertaining to individual channel connections. Based upon the type of signal received, this module forwards the connection signal to the corresponding SAAL for processing. This module then transitions back to the **dispatch** state.

c. ATM_layer Module

The ATM_layer module is the first of the four ATM modules in OPNET. (Together these four ATM modules perform all of the functions of the ATM layer as defined in chapter II.) This module is used in all nodes throughout the network. This module is responsible for coordinating the functions of the other three ATM modules. Primarily, it serves as an interface between the ATM modules, AAL module, and the application modules. It forwards call requests to the ATM_mgmt module for Virtual Path Identifier (VPI) and Virtual Channel Identifier (VCI) assignment. It forwards data cells to the ATM_switch module. The ATM_layer process state diagram is shown in Figure 33. A description of these states is given below.

- **INIT**. When in this state, the module initializes local variables.

- **config**. This state's functionality is identical to the **config** state in the call_generator module on page 70.

- **wait**. This state waits for an event (reception of: a call request, data packets or network messages) to occur and uses it type to determine the next state.

- **AAL int**. This state forwards the call descriptor Ici to the ATM_mgmt module when a call request is received from the AAL module.

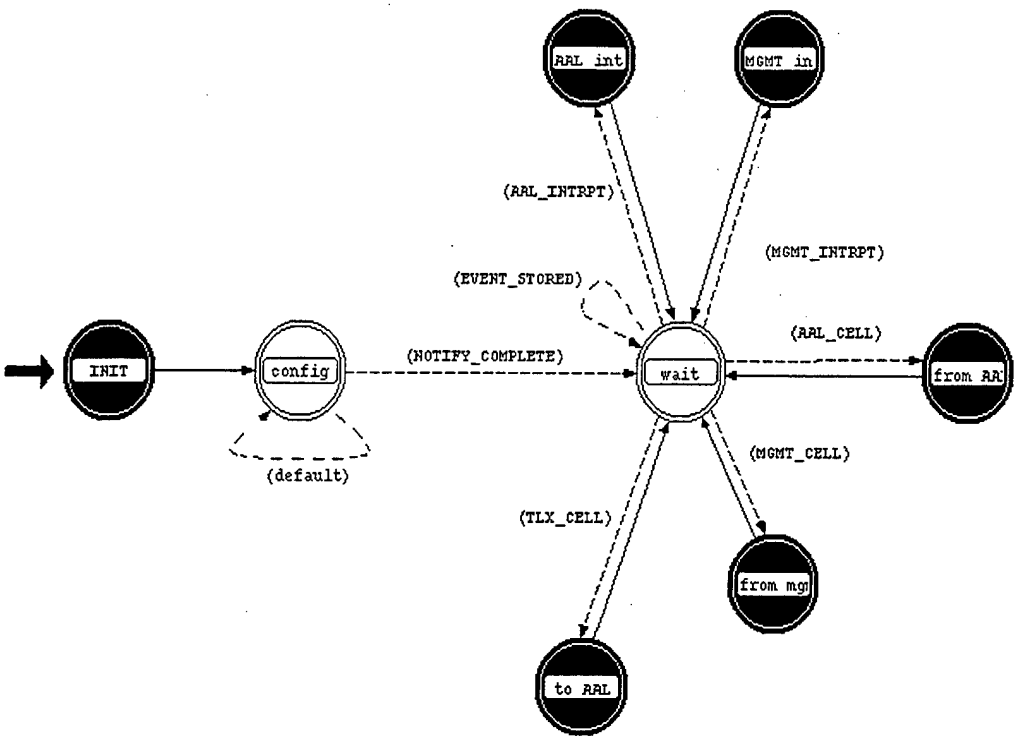


Figure 33. State diagram for ATM Layer module.

- **MGMT int.** This state forwards call descriptor information, indicating the status of a call request, to the AAL module.
- **from AAL.** When a 48-byte data segment arrives from the AAL module, this state is entered and the module adds the appropriate cell header, inserts the data segment into the cell payload, and forwards the newly created ATM cell to the ATM_switch module for transmission.
- **from mgmt.** A process in this state forwards an ATM management cell to the ATM_switch module for transmission.
- **to AAL.** When in this state, the module processes all data cells addressed to this node. After removing the ATM cell header, this state forwards the 48-byte data segment to the AAL module.

d. ATM_mgmt Module

The ATM_mgmt module manages the Virtual Paths (VPs) and Virtual Channels (VCs) for all calls associated with a node. This module is used in all nodes

throughout the network. For a node that originates calls, this module sends and coordinates the call setup and connect messaging. For an intermediary node between the source and destination, this node assigns a VP and VC and then forwards the setup message on to the next node enroute to the destination. For a destination node, the ATM_mgmt module forwards a call request to the ATM_layer and awaits a response from the ATM_layer indicating acceptance or rejection of the call. Based on the acceptance or rejection, this module then sends the appropriate message, corresponding to the acceptance or rejection, back to the source node. The ATM_mgmt process state diagram is shown in Figure 34. A description of these states is given below.

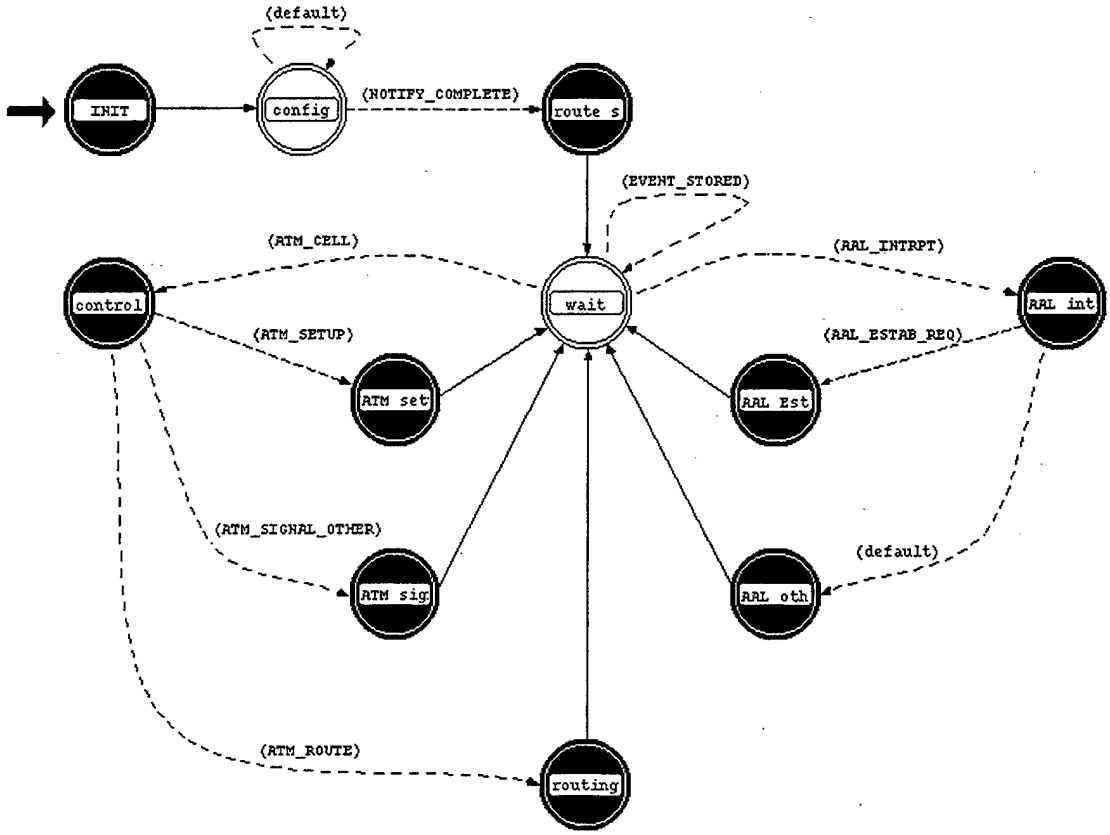


Figure 34. State diagram for ATM Mgmt module.

- **INIT, config, and wait.** These states provide the same functions in support of this network module as the identically named states defined in ATM_layer module on page 73.

- **route start**. In this state, the module spawns a dynamic process to execute dynamic routing. The routing process creates a routing table from this node to all other nodes within the network. (A full description of the dynamic routing process is given later in this section.)

- **AAL intrpt**. This state is used to parse the AAL interrupt type.

- **AAL Estab**. When a call request arrives, this state is used to process the AAL establishment request from the AAL module. This state spawns a dynamic process called **call_src** which then handles this call request. (A full description of the dynamic call_src process is given later in this section.)

- **AAL other**. This state handles other AAL module requests for this node by forwarding it to the appropriate dynamic call_src process (which was previously created in an **AAL Estab** state that corresponds to this request).

- **control cell**. After receiving a control cell from the network, this state is used to determine the type of control cell.

- **ATM setup**. For a call establishment request, this state is entered and the module checks the destination of the call. If this node is the destination, this state is used to spawn a dynamic process, **call_dst**, which then handles this call request. If this is not the destination node, this state spawns a dynamic process called **call_net** which forwards the call request on to the next node. (A full description of the dynamic call_dst and call_net processes are given later in this section.)

- **ATM signal other**. In this state, the module handles other ATM control cells sent to this node. If a dynamic call_net or call_dst already exists to handle the request, this state forwards it to the appropriate process. If the signal is a network call release request, this state is used to forward the release request to the next node enroute to the destination address.

- **routing cell**. For dynamic routing cells, a module in this state forwards the routing message to the dynamic routing process for this module.

e. ATM_trans Module

The ATM_trans module handles all cells received from the ATM network. This module is used in all nodes throughout the network. It first determines the type of an ATM cell: control or data. It forwards the control cells to the ATM_mgmt module. For data cells, this module determines whether the data cell is destined for this node. If the data cells is destined for this node, the module forwards it to the

ATM_layer module, otherwise this module forwards the data cell to the ATM_switch module. The ATM_trans process state diagram is shown in Figure 35. A description of these states is given below.

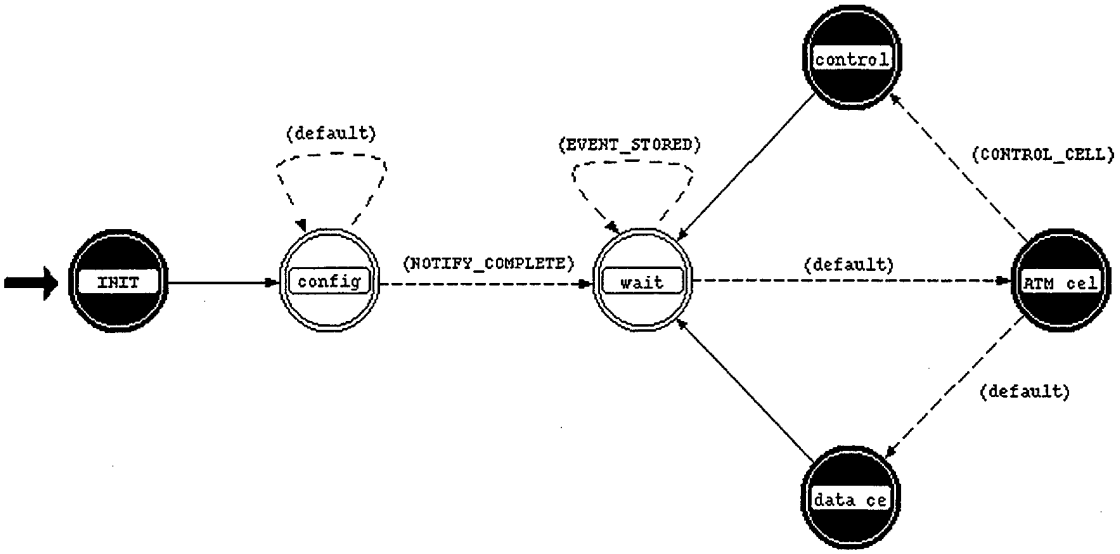


Figure 35. State diagram for ATM_trans module.

- **INIT, config, and wait.** These states provide the same functions in support of this network module as the identical states defined in ATM_layer module on page 73.
- **ATM cell.** This state determines the type of ATM cell that arrived from another node in the network.
- **control cell.** This module transitions to this state when a control cell arrives. It forwards the control cell to the ATM_mgmt module.
- **data cell.** This state is reached upon reception of a data cell. This module delivers the data cells addressed to this node to the ATM_layer module. (For data cells addressed to other nodes in the network, this module delivers the data cell to the ATM_switch module.)

f. ATM_switch Module

This module forwards ATM cells to the next node in the ATM network. This module is used in all nodes throughout the network. The ATM_switch process state diagram is shown in Figure 36. A description of these states is given below:

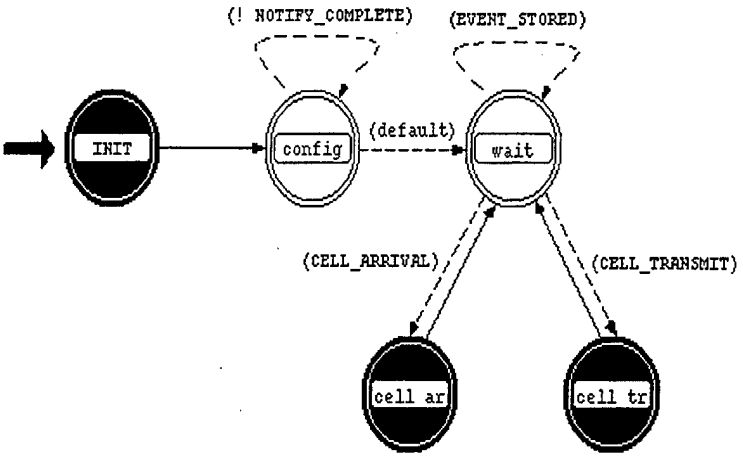


Figure 36. State diagram for ATM_switch module.

- **INIT, config, and wait.** These states provide the same functions in support of this network module as the identical states defined in ATM_layer module on page 73.
- **cell arrival.** In this state, the module processes new ATM cells as they arrive. It places all cells in a queue for transmittal. All cells are held in this queue for an amount of time based on the switching fabric.
- **cell transmit.** In this state, the module dequeues the next ATM cell and sends it to the next node in the network, using a direction that is based on the addressing information contained within the cell.

2. IDS LAN Dynamic Processes

This section describes the dynamic processes required to support the functionality of the modules within the IDS LAN node.

a. SAAL Process

The SAAL process is a dynamic process created by the AAL module to handle the signaling for a call request. It exists in multiple instantiations in the IDS LAN and the TACTICAL LAN, to support concurrent calls in our simulation. Each SAAL process corresponds with a peer SAAL process at the destination node to process the call request. The process state diagram is shown in Figure 37. A description of these states is given below.

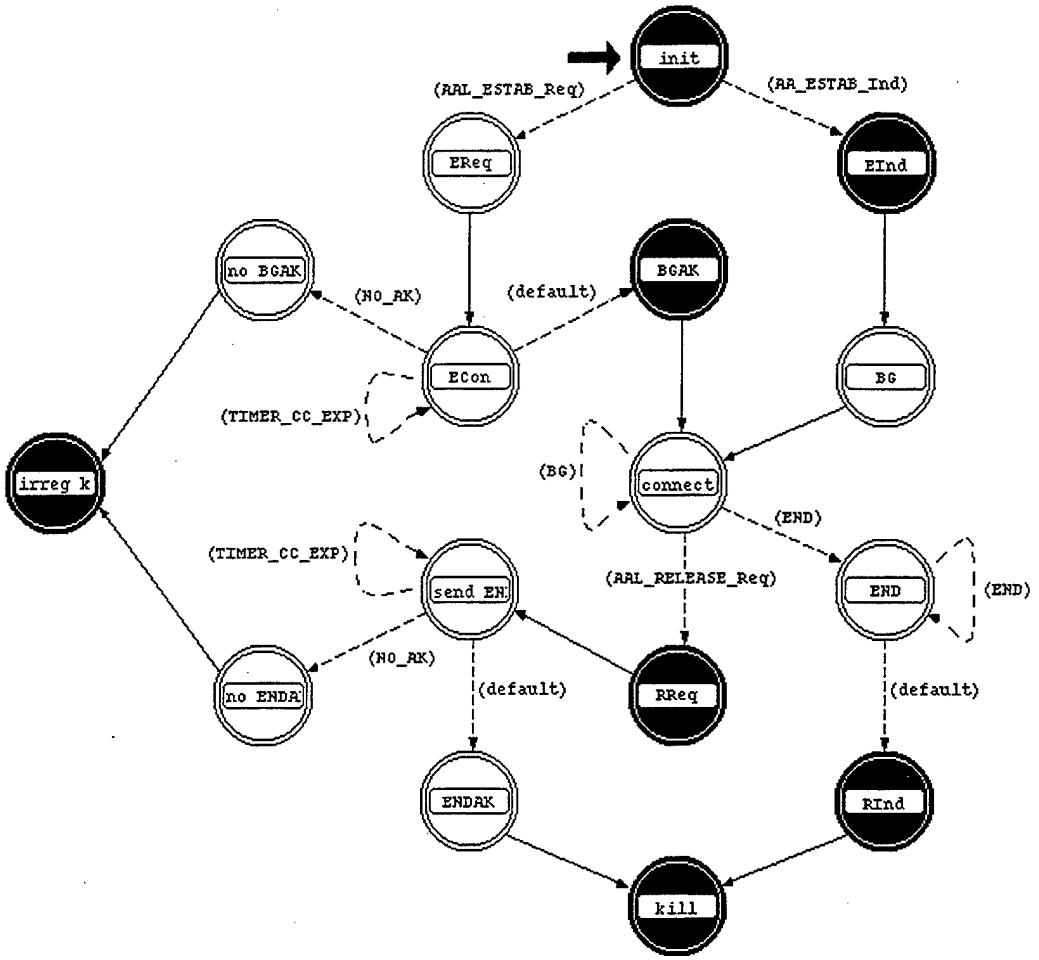


Figure 37. State diagram for SAAL process.

- **INIT**. In addition to code initializing the process's variables, this state's code determines the type of request for which it was spawned.
- **EReq**. This state is used to forward the call requests from the application module to the ATM_layer module. The process remains in this state awaiting a response. If the ATM_layer responds with a call release message, this state's code sends the application module a message canceling the call request and then terminates this SAAL process. If the ATM_layer responds with a call connection message, this state spawns a dynamic process called **AAL5_Conn** and then invokes the process to handle the AAL5 functions for this call.
- **ECon**. This is used to wait for the network to acknowledge the request to establish the connection.

- **BGAK**. When the network accepts a request by sending an acknowledgment, the process in this state sends a message to the application module to start sending data packets. If the network rejects the request, then it sends the application module a message canceling the call request and then terminates this SAAL process.

- **connect**. After establishing the connection, the process waits in this state until either the call completes or the network abnormally terminates the call.

- **END**. When a call successfully completes, the SAAL process in this state sends an END request to its peer SAAL process.

- **RInd**. After sending an END request to its peer, the process enters the **RInd**, where it awaits the response from the network, acknowledging the END request. The acknowledgment indicates the network has freed all resources associated with the call request.

- **kill**. The process in this state destroys the dynamic AAL5_conn process and this SAAL process.

- **EInd**. The SAAL process, at the destination node, enters this state when the call request is delivered from the ATM_layer module. In this state the process handles the request by forwarding the request to the application module. The process also creates a dynamic AAL5_conn process to handle the call request.

- **BG**. The SAAL process waits in this state for a response from the application module. If the application module accepts the call request, the code in this state generates an acknowledgment for the call source node. If the application module rejects the request, the process destroys the dynamic AAL5_conn process and this SAAL process.

- **no BGAK, irreg kill, send END, no ENDAK, ENDAK, and RREQ**. The SAAL process in these states handles those instances when the network sends a negative acknowledgment to a request or fails to send any type of acknowledgment.

b. AAL5_Conn Process

The SAAL process creates and invokes an AAL5_conn process to handle packet segmentation and packet reassembly functions. When spawned by a sender, this process breaks data packets received from the application layer into 48-byte segments and forwards the segments to the ATM_layer module. When spawned by the receiver, it reassembles the 48-byte data segments from the ATM_layer module into

a data packets. The process state diagram is shown in Figure 38. A description of these states is given below.

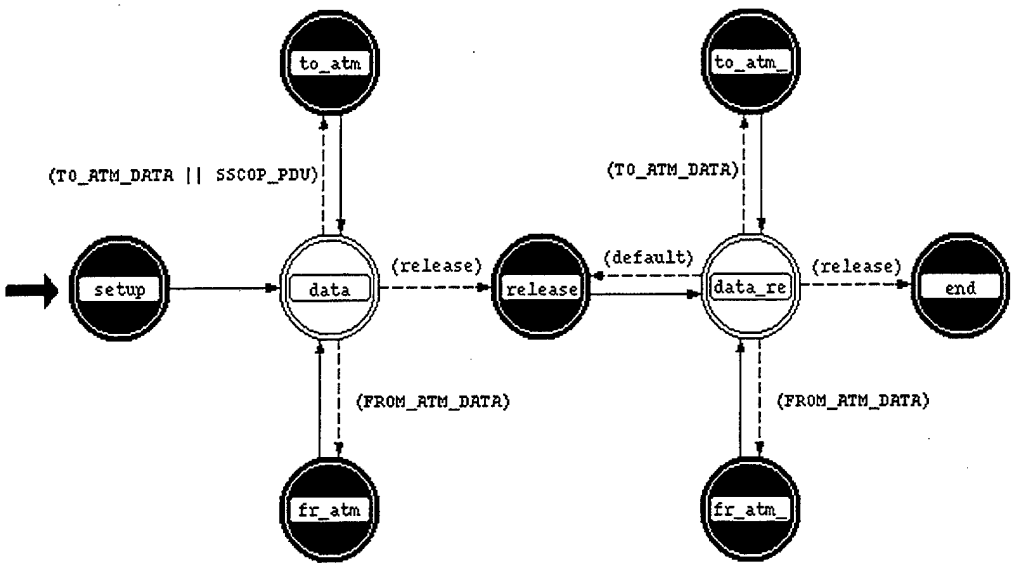


Figure 38. State diagram for AAL5 Connection process.

- **setup**. When in this state, the process initializes local variables and creates the buffers used for segmentation and reassembly.
- **data**. When data arrives, the process enters this state and determines the type and source of the data. For call requests and call data packets from the application module, the process transitions to the **to_atm** state. For data segments from the ATM_layer module, it transitions to the **from_atm** state.
- **to_atm**. In this state, the AAL5_conn process decomposes messages into 48-byte segments and sends the segments to the ATM_layer module.
- **fr_atm**. When the process enters this state, it reassembles the data segments into a data packet and forwards the packet to the application module.
- **release**. In this state the process decomposes a call completion message from the application layer into 48-byte segments and sends the segments to the ATM_layer module.
- **data rel**. After starting the connection release process, no more data packets can be sent but additional data segments are still accepted from the ATM_layer. This process transitions to the **to_atm_rel** state when call data packets arrive

and transitions to the **from_atm** state when data segments from the ATM_layer module arrive.

- **to_atm_rel**. This state is used to destroy all application data packets received after sending the connection release message.
- **from_atm_rel**. In this state, the process continues to accept additional ATM data segments until the connection terminates.
- **end**. This state is used to destroy the AAL5_conn process.

c. Routing Process

The ATM_mgmt process creates and invokes a routing process to handle call routing for the ATM node. This process constructs and maintains the dynamic routing table. The routing table uses an adaptation of the Bellman-Ford Shortest Path algorithm [Ref. 16, 4]. The process state diagram is shown in Figure 39. A description of these states is given below.

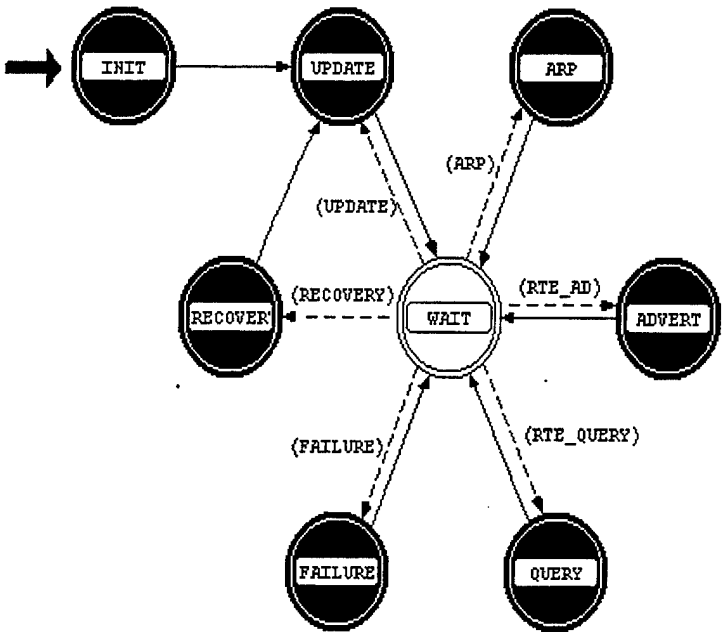


Figure 39. State diagram for Dynamic Routing process.

- **INIT**. The code in this state initializes the local variables and creates an empty routing table.

- **UPDATE**. Periodically, the routing table is checked for accuracy. In this state, the routing process removes any old links from the routing table and recalculates the cost for all current routes. The code in this state also broadcast the address resolution packets (ARP) to all neighboring nodes. The ARP ensures that all neighbors maintain active routes containing information about the current node.

- **WAIT**. When in this state, the process waits for an event to occur.

- **ARP**. The process transitions to this state when address resolution packets arrive. These packets are used to verify and update the routing table as necessary.

- **ADVERT**. The process transitions to this state when route advertisement packets arrive. The process verifies that the route contained in the route advertisement packet is valid and that the route does not cause a loop in the network. The process then updates, using the new route, the cost of all routes in the routing table.

- **QUERY**. When in this state, another process has requested the shortest route to a particular destination node. The code in this state searches the routing table to determine the best route available and verifies that the route is still available. It returns the output port number for the best route to the destination node.

- **FAILURE**. Because a link failed in the network, this process must update the routing table by removing the link and calculating a new cost for routes previously using the failed link. Additionally, the code within this state broadcasts a route advertisement packet to all neighboring nodes notifying them of the new cost of all routes through the current node.

- **RECOVERY**. During a simulation, nodes may be programmed to temporarily fail and then recover from the failure. When failure occurs, OPNET issues a recovery interrupt. The process transitions to this state to handle the recovery interrupt and it then transitions to the **UPDATE** state.

d. Call_Src Process

The ATM_mgmt process, at the calling node, creates and invokes a call_src process for each VCC to handle ATM signaling (VCC setup and release). This process is then destroyed at connection termination. The process state diagram is shown in Figure 40. A description of these states is given below.

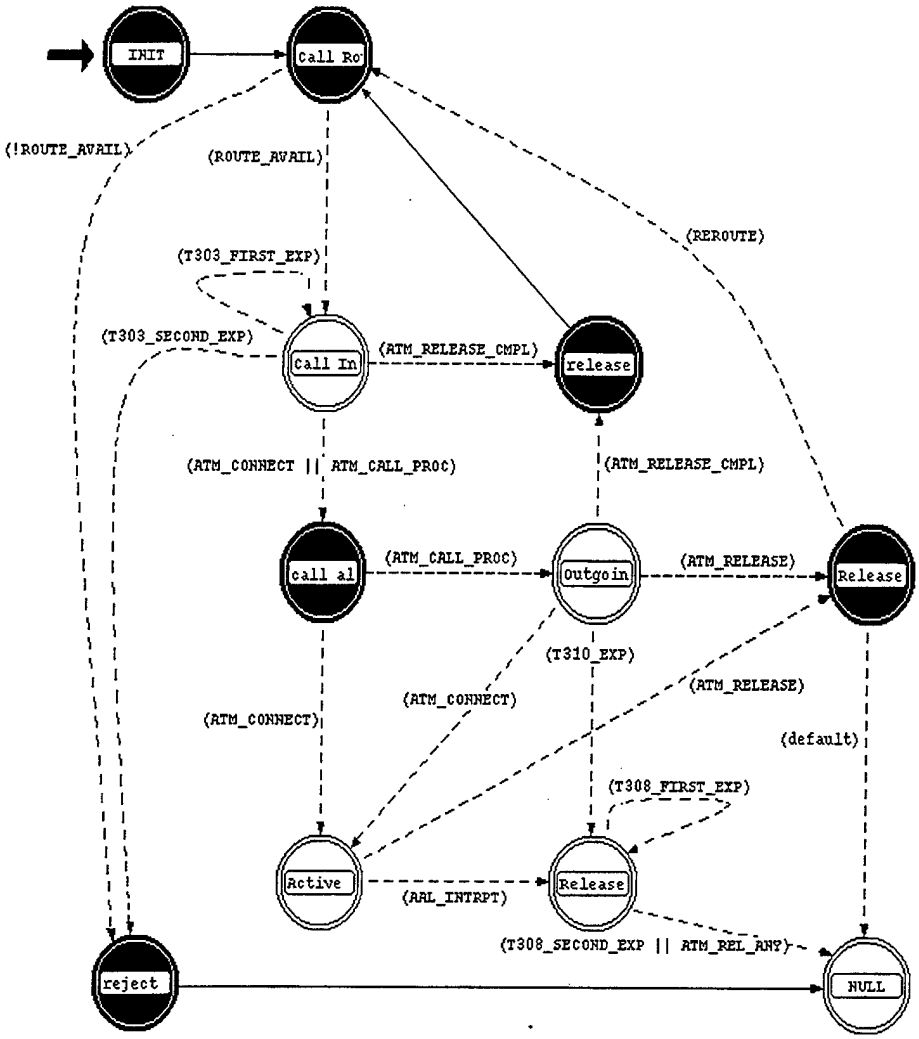


Figure 40. State diagram for Call_Src process.

- **INIT**. This state is responsible for initializing numerous variables within the process model. Additionally, this state obtains the call description from the ATM_mgmt module.

- **Call Route**. Code in this state determines whether a route exists from the source node to the destination node and ensures sufficient bandwidth is available on the route. This state invokes the dynamic routing process created by the ATM_mgmt module.

- **Call Initiated**. Here the module sends a setup message to the next node enroute to the destination node. The process then waits in this state for a

response from the network. The network may accept the call, reject the call, or not respond at all.

- **call alloc.** When the network responds with a call accept or call proceeding message, this state's code allocates resources at the source node to support the call request.
- **Outgoing Call Proceeding.** Code in this state waits for the connection request to propagate through the network.
- **Active.** This state is reached when a 'connect' message is received from the network, meaning that the connection is available for data transmission. It then sends the 'connect ack' message to the first intermediate node.
- **Release Request.** When in this state, the module handles the release connection request by first sending the 'release connection' message to the network. After the network releases the connection, it sends a 'release confirm' message to the AAL module.
- **NULL.** Code in this state is responsible for releasing the connection resources at the source node and destroying this call_src process.
- **reject call, release cmpl, and release indication.** These remaining states handle those instances when the network rejects a connection request or simply fails to respond to a connection request.

e. Call_Net Process

The ATM_mgmt process, at each intermediate node, creates and invokes a call_net process for each VCC to handle ATM signaling (VCC setup and release). This process is destroyed at connection termination. The process state diagram is shown in Figure 41. A description of these states is given below.

- **INIT.** In the INIT state numerous variables within the process model are initialized. Additionally, this state determines whether the connection request previously passed through this node.
- **No Fwd Call.** If this state is reached we know that the connection request previously passed through this node, and consequently the code destroys both the request and this process.
- **Call Initiated.** Here the module determines whether a VP and a VC are available to support the connection.

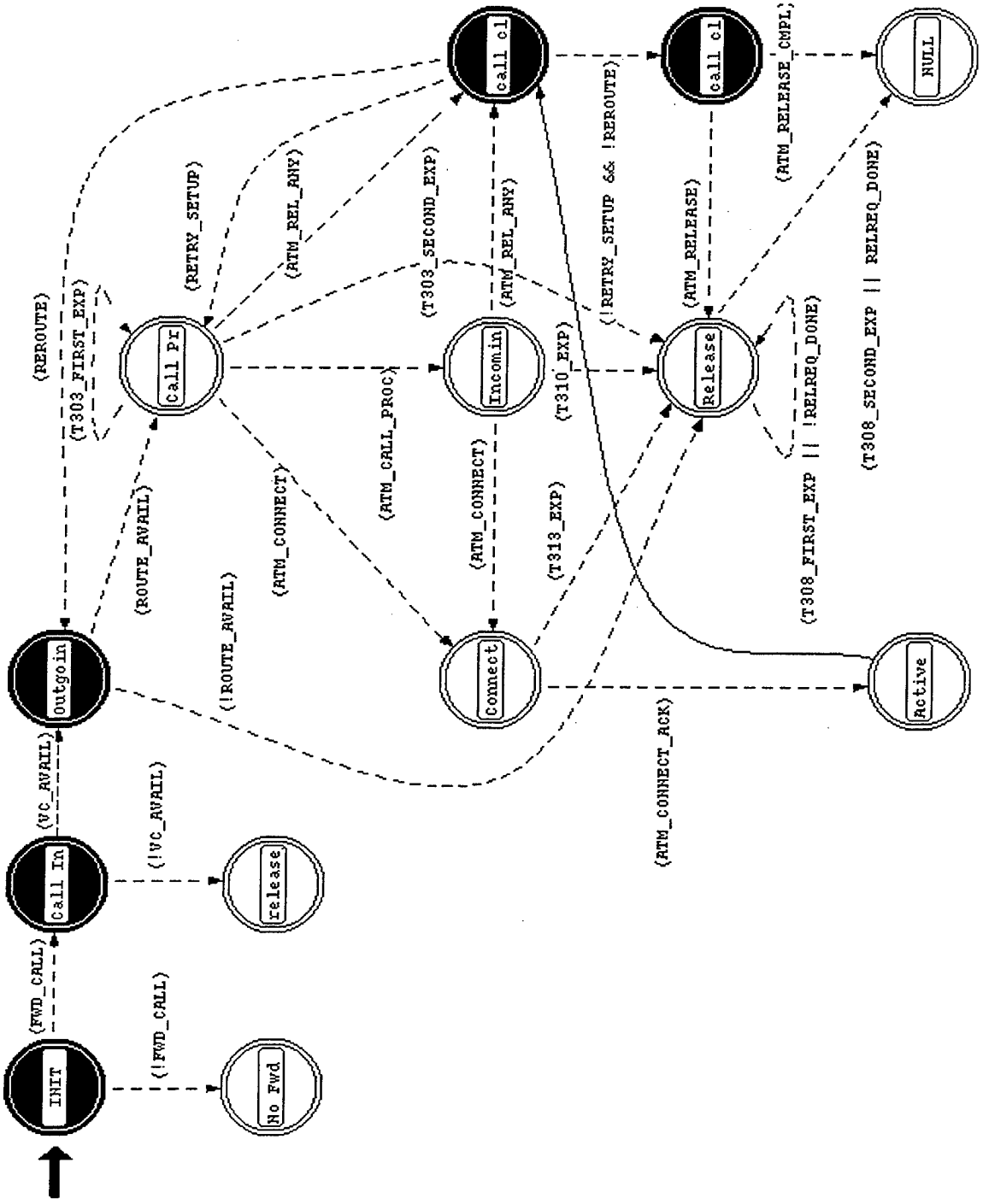


Figure 41. State diagram for Call_Net process.

- **release cmpl.** When resources are not available to support the connection request, this state is used to send a connection rejection message to the source node after which it destroys this process.
- **Outgoing Call Proceeding.** When resources are available to support the connection request, this state is used to assign resources to the connection.
- **Call Present.** This state is used to send a setup message to the next node enroute to the destination node. The process then waits for a network response. The network may accept the connection, reject the connection, or not respond at all.
- **Connect Request.** When the network accepts the connection, this state is used to forward the acceptance towards the source. The module also sends a connection acknowledgment to the previous node.
- **Active.** Reaching this state means that the connection is available for data transmission. The process then remains in this state awaiting a signal to release the connection.
- **Release Indication.** When this state is reached, the network has rejected the call or failed to respond to the connection request. The process then sends a call release message to all the other nodes within the network.
- **call clear and call clear cont.** These states are used to free resources allocated to a connection and to send a release message to other nodes within the network.
- **NULL.** This state destroys this call_net process.

f. Call_Dst Process

The ATM_mgmt process, at the destination node, creates and invokes a call_net process for each VCC to handle ATM signaling (VCC setup and release). This process is destroyed at connection termination. The process state diagram is shown in Figure 42. A description of these states is given below.

- **INIT.** This state is used to initialize numerous variables within the process model.
- **Call Init.** After the arrival of a connection request, a process in this state determines whether resources are available to support the call.

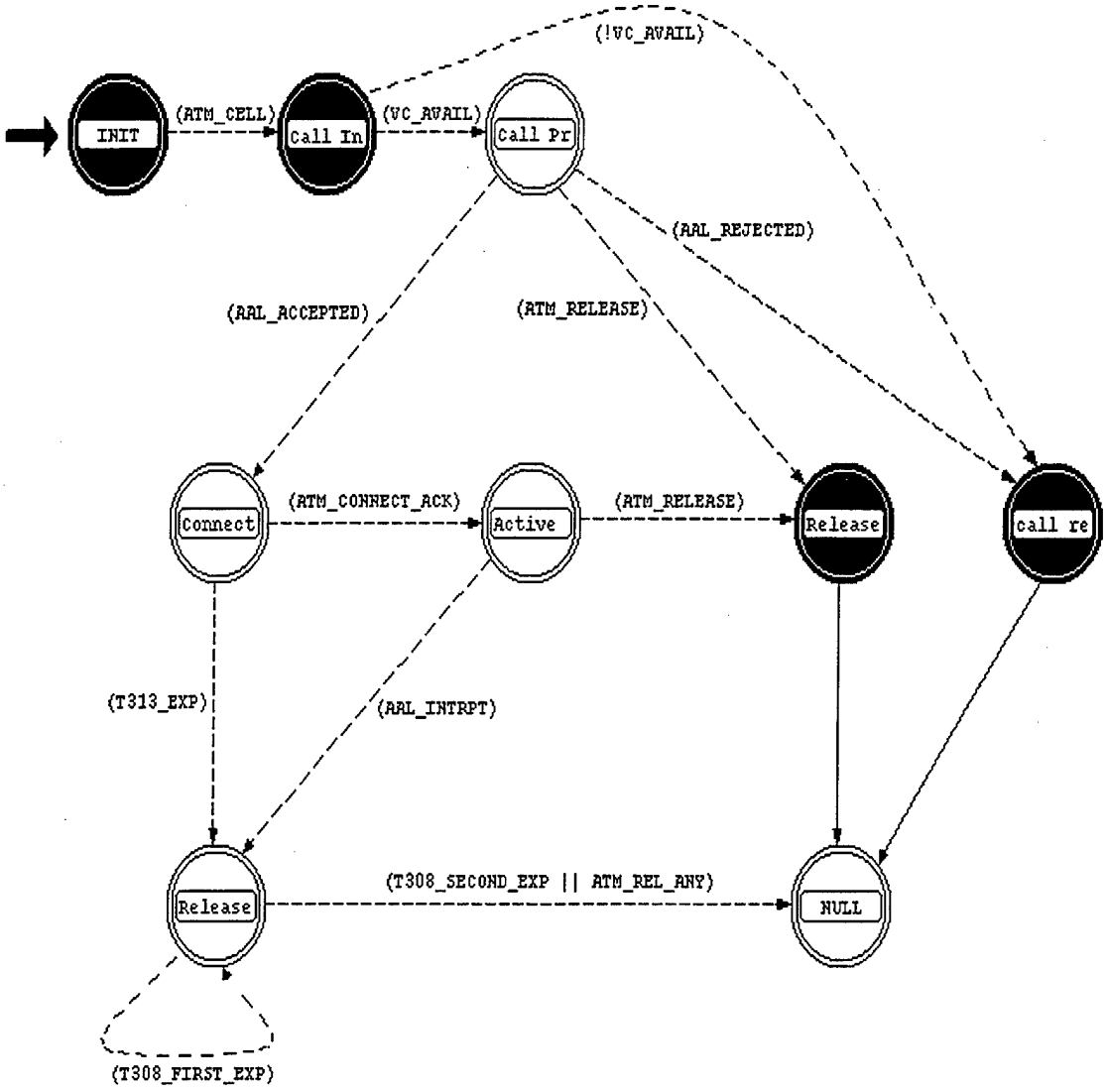


Figure 42. State diagram for Call_Dst process.

- **Call Present**. When resources are available this state is used to allocate the necessary resources and send the call request to the AAL module at the destination node. Additionally, it sends a call proceeding message to the previous intermediate node in the network.

- **Connect**. If the AAL accepts the call request, this state is used to send the connect message to the intermediate node in the network.

- **Active**. This state is reached when the connection is available for data transmission. The process waits in this state for the AAL module to signal call completion or for the network to drop the connection. If the AAL module signals call completion, the next state is the **Release Request** state. If the network dropped the connection, the transition is to the **Release Indication** state.

- **Release Request**. This state is used to send a release connection message to other nodes in the network and waits for a response from the network.

- **Release Indication**. A process in this state sends a release acknowledgment message to the previous node in the network.

- **call rejected**. This state is used to send the connection rejection message to other nodes in the network.

- **NULL**. This state is used to return resources that were allocated to the connection and to destroy the call_dst process.

3. Uplink LDR and Dnlink LDR

For our network, we chose to represent the Uplink LDR and Dnlink LDR as simple ATM switches. Figure 43 shows the node diagram for our LDR switch. This node contains the four ATM modules that we discussed in the previous section. Unlike the IDS LAN these nodes support three communication links rather than one. The three transceiver links are represented by the following modules: (pr_0, pt_0), (pr_1, pt_1), and (pr_2,pt_2). Two of these links support a transmission channel capacity of 155 Mbps. The third supports only an 8 Mbps channel capacity. The low data rate channel represents the BADD ATM channel capacity available through the GBS system.

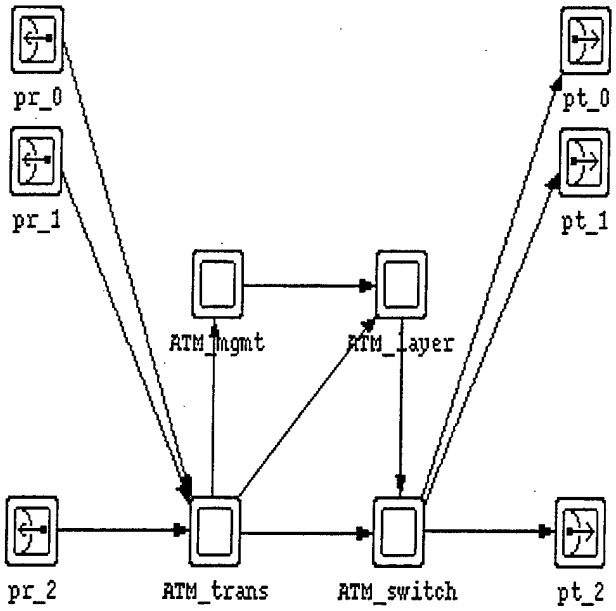


Figure 43. Uplink and Dnlink Low Data Rate switch node diagram.

4. Tactical LAN Subnetwork

The Tactical LAN subnet in our model contains a single WFA switch and the traffic destination as shown in Figure 44. Our model is extensible in that it would require very little modification to add additional WFA switches and traffic destination modules.

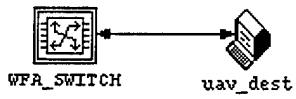


Figure 44. Tactical LAN Subnetwork diagram.

a. WFA Switch

The WFA switch is similar to the Uplink LDR switch shown in Figure 43. It contains the four ATM modules and two transceivers, both with a channel capacity of 155 Mbps.

b. Traffic Destination

The traffic destination node is similar to our traffic source in Figure 30. The only difference is that the call_generator has been replaced with a traffic sink. As data segments arrive at the destination node they are reassembled by the AAL module and forwarded to the traffic sink. The traffic sink updates requested statistics and then destroys the data packet.

5. Network Communication Links

After defining the nodes within the network, we connected the nodes by using OPNET communication links. As mentioned earlier in this section, OPNET currently does not support simplex ATM channels, therefore we first inserted the standard OPNET duplex communication links, then modified the code.

Our model utilizes two types of communication links: one with a capacity of 155 Mbps and the other with a capacity of 8 Mbps. The standard communication link with 155 Mbps capacity connects the traffic source with the Uplink LDR, the Dnlink LDR with the WFA switch, and the WFA switch with the traffic destination. We modeled this link after a standard fiber optic connection with no errors and no delay.

For the backbone connection between the Uplink LDR and the Dnlink LDR, we created a link with a capacity of 8 Mbps using the OPNET parameter editor described in Chapter V. Initially, we defined the communication link with a transmission delay of 0.25 seconds which is a typical delay time for a single geosynchronous satellite transmission.

6. Modification of Our Initial OPNET Modeler

After constructing our initial OPNET model, we began testing to verify that the network was being correctly simulated. When we ran simulations with the link delay set to 0.25 seconds, the simulation terminated abnormally giving the error message 'traffic_dest received process handle for destroyed process.' Tracing the code, we found a simulation timer called AMSC_AAL_TIMER_CC_DURATION in

the ams_aal_interfaces.h file. This timer defines the maximum amount of time, in seconds, that an AAL process can wait for a response to the connection request message. This variable was originally defined as 0.25 seconds; we changed the timer to 10.0 seconds and the simulation ran successfully.

Another problem that we encountered in the testing of our basic network model was the calculation of **Peak Cell Rate** (PCR) for small calls. When the packet size is less than 384 bits, OPNET calculates the PCR to be 0. Again, tracing the code we found the problem to be integer division. We modified the OPNET code to explicitly cast the calculation as a float, which yields a correct PCR.

C. STATIC CHANNELS WITHIN BADD ATM NETWORK

The next phase of our network design required the definition of static channels within the BADD ATM network. Chapter II describes the call setup protocol for virtual channel definition within ATM. Normally virtual channels are created and destroyed dynamically by the network. Because the GBS backbone within BADD only allows simplex communication, BADD's designers chose to define static channels for the Uplink LDR and Dnlink LDR during network configuration.

To simulate this configuration, we modified our network in two phases. In the first phase we defined static channels at each ATM switch within the network and in the second phase we modified the network to use the static channels.

For phase one, we modified the ATM_mgmt module, described on page 74, by adding a state called **static_channels_setup**. The new ATM_mgmt module is shown in Figure 45.

The code for the newly added static_channels_setup state begins by opening the input_channels file and reading the first line. The first line contains a single integer value defining the number of static channels for the network. The maximum allowable channels are 1024 [Ref. 10]. Each subsequent line in the input_channels file defines a single channel by giving its bandwidth capabilities. This state is then used to add

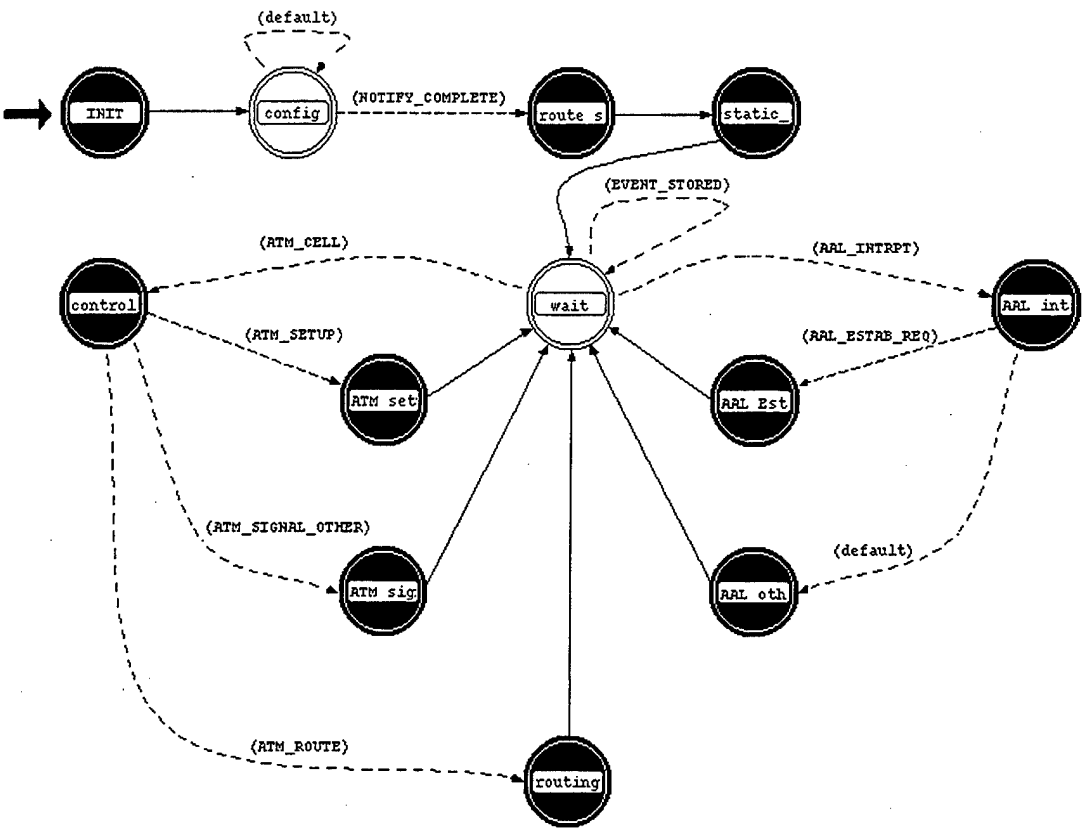


Figure 45. State diagram for BADD ATM Mgmt module.

the static channels to each ATM path and allocate the bandwidth to the channels as defined in the input_channels file.

The second phase requires the modification of the call_src, call_net and call_dst modules. Each of these modules called functions for creating and destroying virtual channels as described on pages 83 through 87. We created new functions for allocating only the static channels and deallocating the channels. In the call_src module, we modified the states **call route**, **release cmpl**, **call alloc**, **release indication**, and **NULL** to call our new functions. In the call_net module, we similarly modified the states **call initiated**, **outgoing call processing**, and **call clear** to invoke our new functions. Similarly, in the call_dst module, we modified the code for the states **Call Init**, **Call Present**, and **NULL**. Appendix B contains the code for these

modifications.

Although the simulation ran successfully, link utilization was extremely low as shown in Figure 46. We found that the network spent the majority of time sending ATM control data to setup and tear down a connection. Although this overhead is typical for duplex, dynamic channels, it should be absent in simplex, static virtual channels.

According to OPNET Communication Mechanisms, any delay defined on a communication link is applied equally to all data traversing the link [Ref. 4]. This means that all ATM cells, including both data and control cells, traversing the GBS backbone are delayed for 0.25 seconds. This application of delay is not appropriate for the BADD network because the link is simplex with static virtual channels. Static channels on the satellite link are defined at the switches and there is no need for signaling. In addition, since the satellite link is simplex there is no signaling coming back from the downlink to the uplink. Since we were using the OPNET duplex implementation as a baseline, we resolved these conflicts by setting the communication link delay to 0.0 seconds and rewriting the ATM_switch module to send only data cells with a 0.25 second delay over the GBS backbone. This modification triples the link utilization as illustrated in Figure 47.

D. FINAL BADD TRAFFIC SOURCE

In the initial design we described only a single call generator. In this section we describe our experiences when adding additional call generators. As we increased the number of call generators we observed that calls were being dropped. ATM normally drops call requests when sufficient resources are unavailable to handle the call requested. When a rejection was sent to our call generator, it would respond by immediately issuing another request. This looping process lead to a dramatic increase in the number of call requests and call rejections. This problem lead us to redesign our traffic source node as depicted in Figure 48. This section describes the function

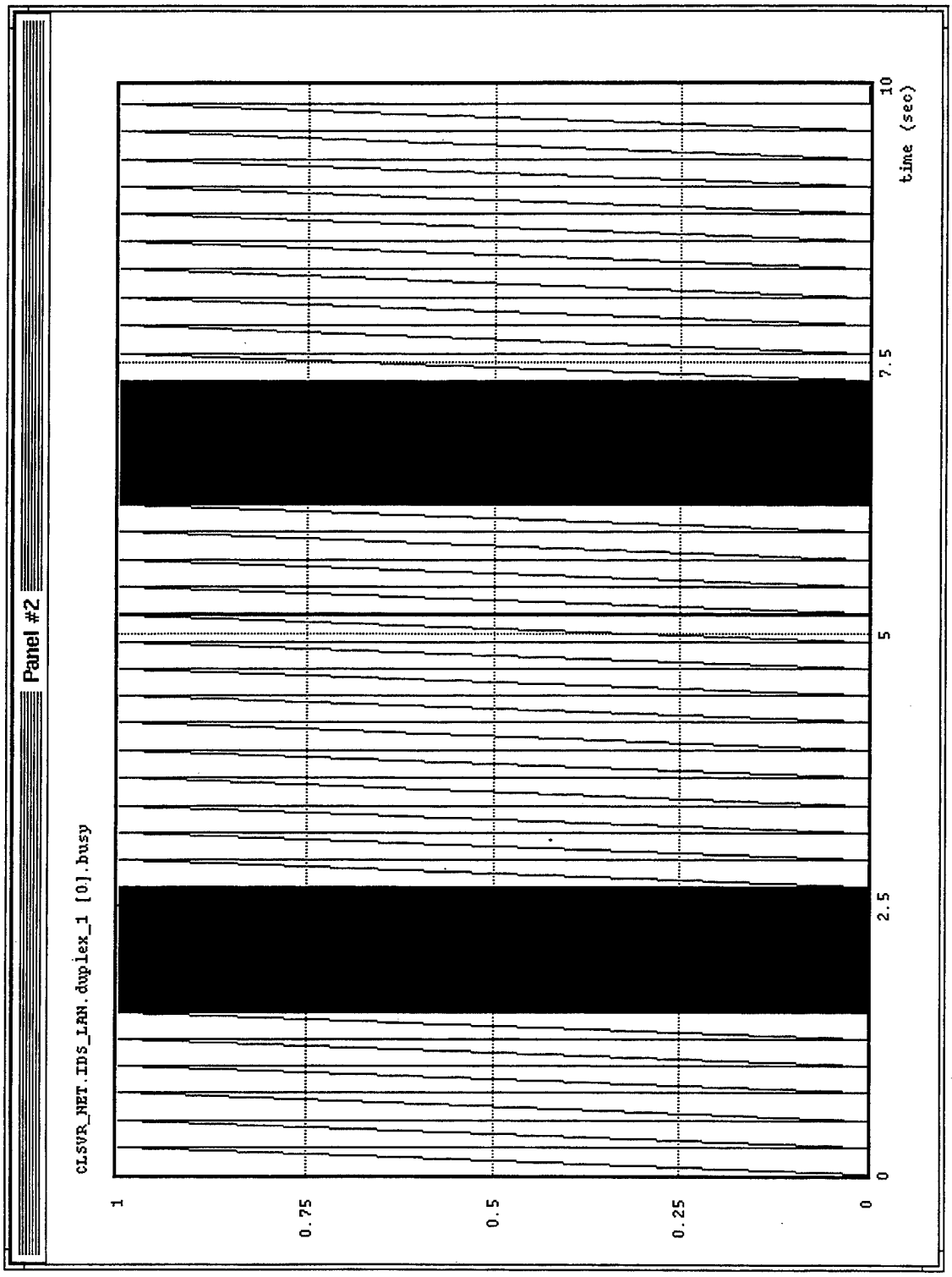


Figure 46. Initial link utilization for our basic BADD network.

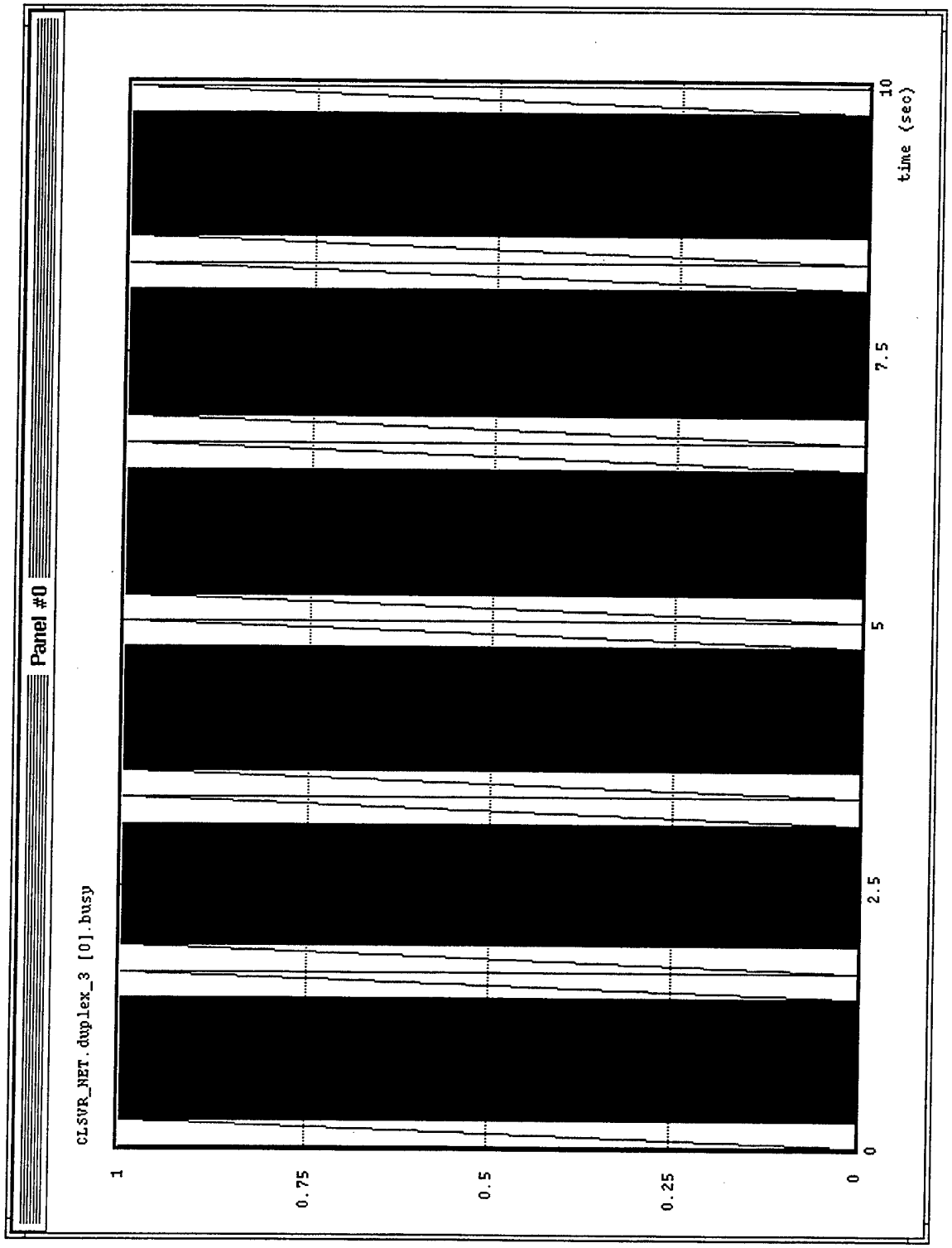


Figure 47. Link utilization after modification of our basic BADD network.

of our modified call_requester module, our new call_scheduler module, and our new dynamic process called call_generator.

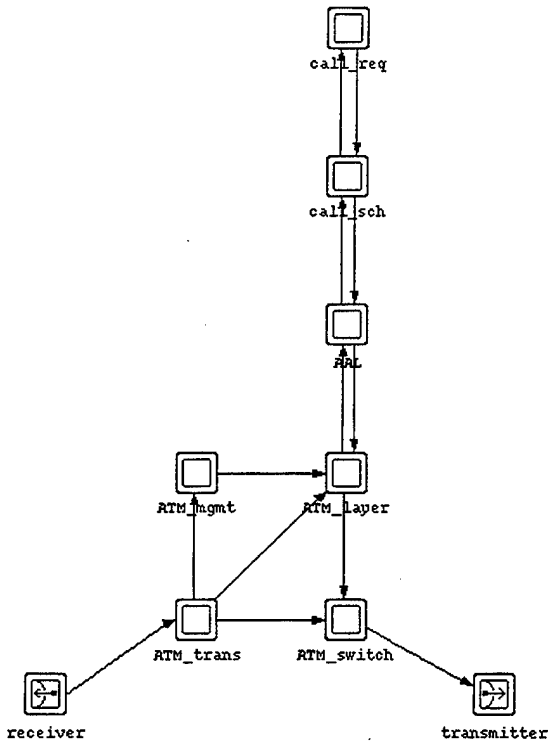


Figure 48. Node Diagram for our new BADD Traffic Source.

1. BADD Call Requester

OPNET's call_generator module generates a single call, waiting until the completion of the previous call before starting the next call. When we began queuing calls to await resources, the call_generator process would not generate another call until the call had been removed from the queue and processed. This wait time does not represent a realistic scenario, i.e., a database server does not wait for an acknowledgment indicating that its last response was sent to the final destination before processing the next query. To correct this problem, we divided the call generation process into two distinct processes: call request generation and data generation.

Figure 49 shows our new call_requester module. This process iteratively generates a call request and then, using the input mean interarrival rate, generates the event for the time to generate the next call and continues processing regardless of the status of the requested call. A description of the call_requester states is given below.

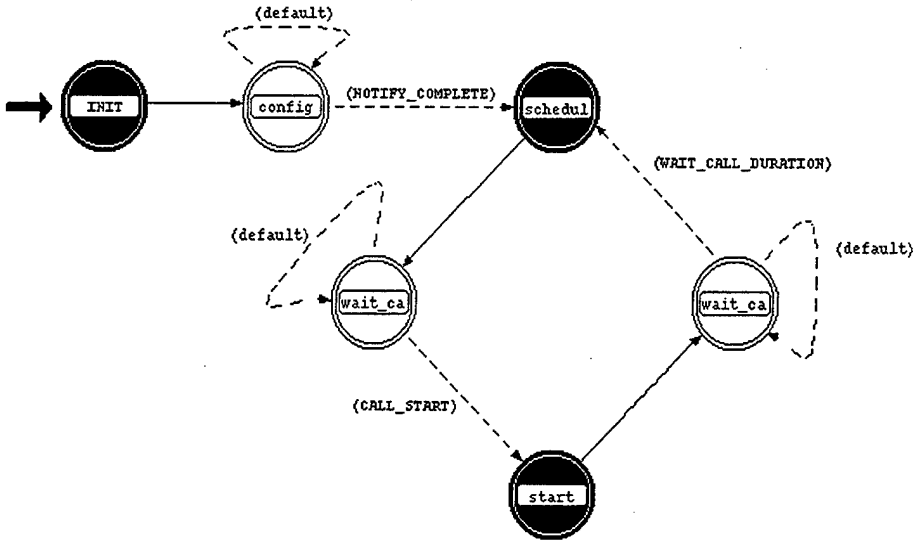


Figure 49. State diagram for BADD call_requester module.

- **INIT, config, schedule, and wait.** These states provide the same functions in support of this network module as the identical states defined in our initial call_generator module on page 70.
- **start.** When the module enters this state, it creates a badd_call_req_if_ici packet describing the call. The module attributes read during the **INIT** state are copied into the Ici packet before it passes the call request to the call_scheduler module. Lastly, this module generates an event, in the future, at which time the module will schedule the next call request.
- **wait_call_duration.** The module remains in the **wait_call_duration** state until the time for the event generated in the previous **start** state arrives. The module then transitions back to the **schedule** state.

2. BADD Call Scheduler

The call_scheduler module accepts call requests and schedules the calls for transmission over the static channels. The state diagram for our new call_scheduler module is shown in Figure 50. A description of the call_scheduler states is given below:

- **INIT**. In this state, the module initializes local variables and reads the model attribute variables. The model attributes, calls pending and scheduler delay, define the maximum number of call requests that will be allowed to be enqueued before they are assigned a time and virtual channel. The scheduler delay is the maximum time that any request can remain in the queue before it is assigned a time and channel. While in this state, the module also creates the global wait queue and the individual static channel queues.

- **config**. This state provides the same functions in support of this network module as the identical state defined in our initial call_generator module on page 70.

- **shared data**. The module transitions to this state after neighbor notification, discussed at page 70 , is completed. The code in this state initializes variables shared by all the processes within the call_scheduler module. The call_scheduler will later spawn a dynamic call_generator process which will access these shared variables to determine the current ATM network configuration.

- **dispatch**. The code in this state functions as a task dispatcher, starting tasks based on the type of interrupt received.

- **call request**. Upon the arrival of a call request, this module transitions from the **dispatch** state to the **call request** state where it inserts the call request into the calls_pending list. The calls_pending list contains all of the calls that require scheduling. After adding the call to the calls_pending list, control returns to the **dispatch** state.

- **call schedule**. If the number of calls in the calls_pending list exceeds the calls_pending attribute, or the scheduler delay attribute timer expires, then the module transitions from the **dispatch** state to the **call schedule** state. Upon entering the **call schedule** state, the code schedules all of the calls in the calls_pending list by assigning each call to a static channel according to one of the scheduling algorithms described in chapter IV. As calls are added to a static channel, the call is time-stamped with the time it was enqueued. When calls are scheduled on an idle channel, the module generates a call_start signaling event that starts the first call on the idle channel. After scheduling all available calls, the module transitions back to the **dispatch** state.

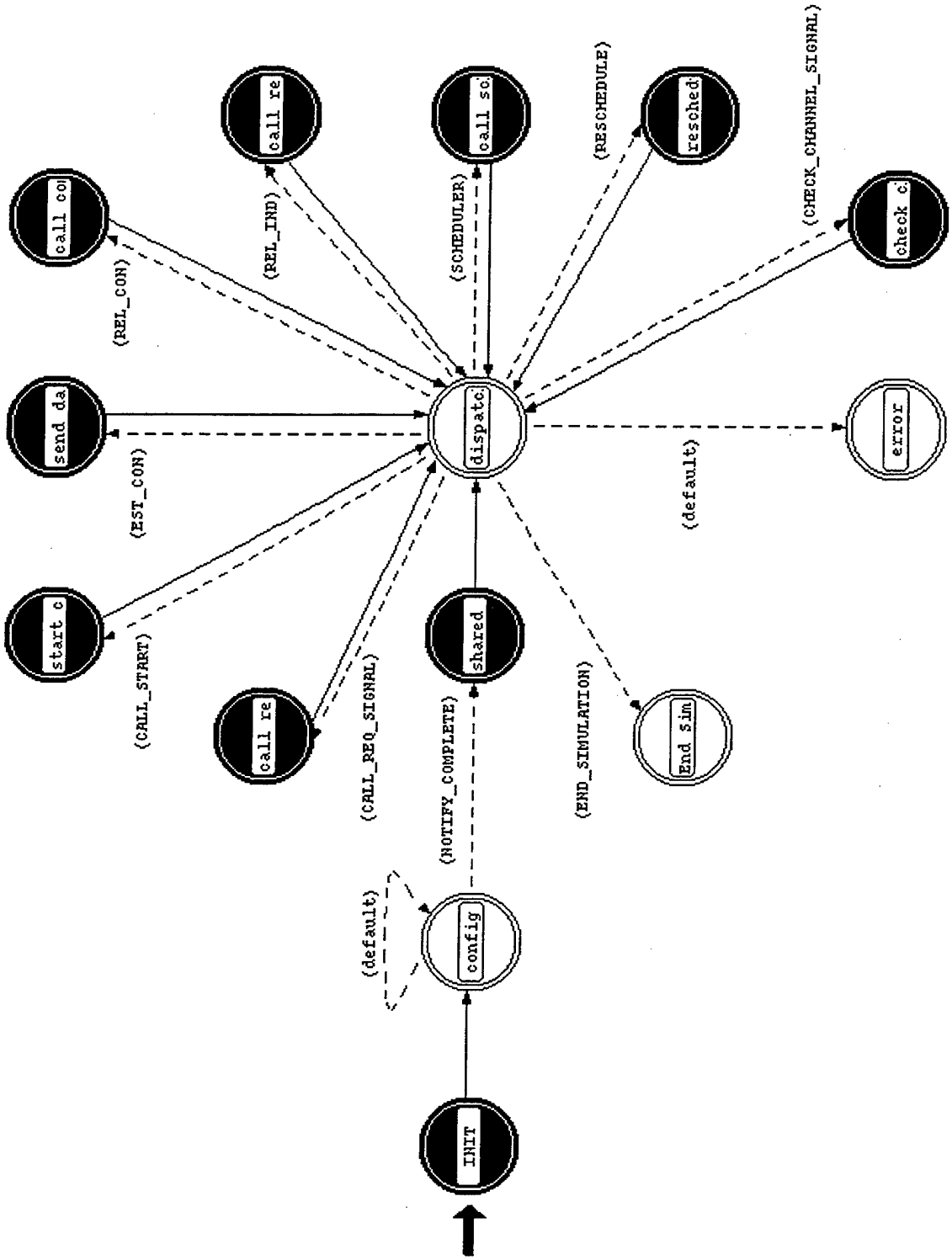


Figure 50. State diagram for BADD call_scheduler process.

- **start call**. When the call_start event arrives, the module transitions to this state. The module dequeues the first call for the channel and spawns a dynamic call_generator process and assigns that process to handle this call. It then transitions back to the **dispatch** state. (The description for call_generator process follows in the next subsection.)

- **send data**. Because all signals for a dynamic call_generator process are returned to the parent call_scheduler module, there must exist a mechanism whereby it can identify which call_generator to invoke. OPNET uses a process pointer, called **prohandle**, to uniquely identify each dynamic process. All signals returned to the call_scheduler contain the prohandle for the required call_generator. Since the call_generator process invoked in the **call start** sends a call connection request to the AAL module, the AAL module signals the call_scheduler when the connection is established. This signal forces a transition from the **dispatch** state to the state **send data** state. This module awakes the sleeping call_generator process and signals the call_generator to start sending data on the established connection. After invoking the call_generator, the module again transitions to the **dispatch** state.

- **call complete**. The call_generator continues to send data until the call completes transmission. The call_generator then signals the AAL module to release and close the connection. The AAL module again signals the call_scheduler after releasing the connection. This release connection signal forces a transition to the **call complete** state. Upon entering this state, the module invokes the requested call_generator and signals the call_generator with a release confirmed acknowledgment. As before, the module transitions to the **dispatch** state.

- **check channel**. When the call_generator receives the signal that the release was confirmed, the call_generator notifies the call_scheduler that the channel is available. This signal forces a transition to the **check channel** state where the channel list is checked for additional calls waiting. If additional calls are scheduled on the channel, the module generates an event to start the next call on the channel.

- **call release**. This state is released when the call_generator sends a call connection request to the AAL module and the AAL module responds with a call rejection. The module invokes the required call_generator and passes along the call rejection signal.

- **reschedule**. When the call_generator receives a call rejection signal, it creates a call reschedule request and signals the call_scheduler to reschedule the call. The signal to reschedule forces a transition to the **reschedule** state which places the call request at the head of the calls_pending list for rescheduling.

- **error**. The code in this state processes unexpected signals.
- **End Sim**. The code in this state prints reports at the end of the simulation.

3. BADD Call Generator

The dynamic call_generator process is created and invoked by the call_scheduler module to manage individual call requests. This process initiates the call connection request, generates data for the call, and releases the call upon its completion. Figure 51 shows our new call_generator process. A description of these states is given below.

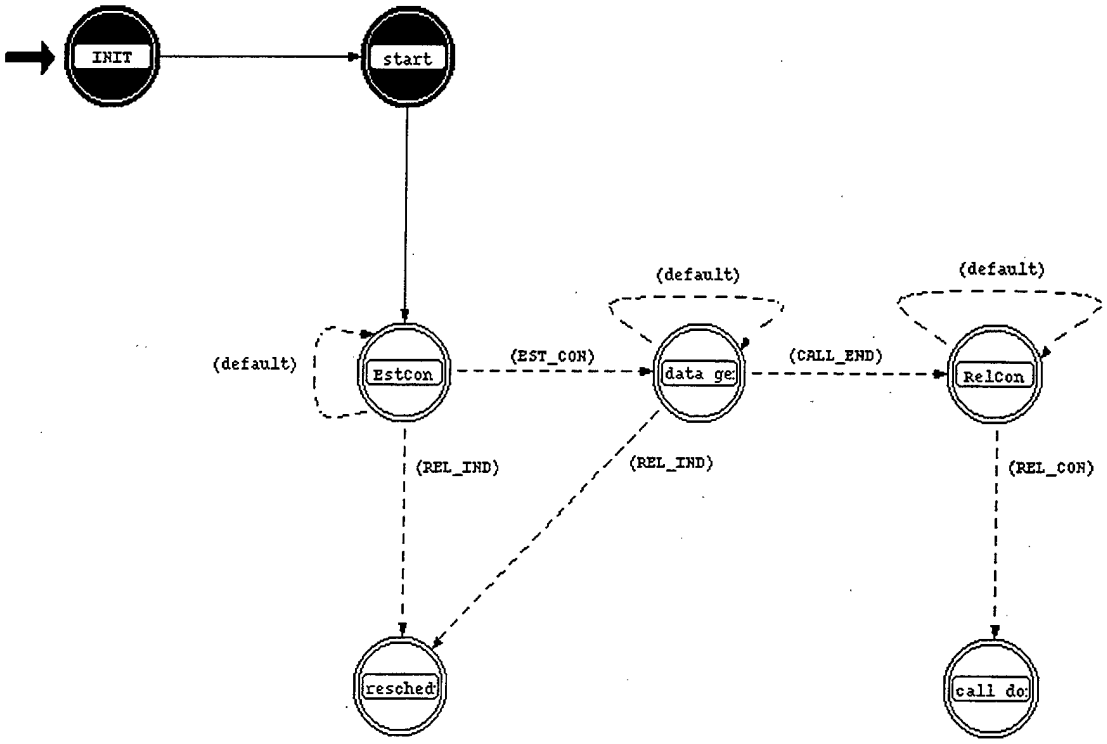


Figure 51. State diagram for BADD call_generator process.

- **INIT**. The call_generator enters this state only upon initial invocation by the call_scheduler module. This state accepts the call request Ici packet and initializes the state variables that describe the call request. Additionally, this state accesses the shared memory initialized by the call_scheduler and retrieves the index identifying the packet stream to the AAL module. All call requests and the call data are sent to the AAL module using this stream index.

- **start**. With the automatic transition to this state, the call_generator creates a badd_call_gen_if_ici packet describing the connection required to support this call. It includes this packet in the call connection request sent to the AAL module. This process then goes to sleep in this state, awaiting a wake-up signal from the call_scheduler module.

- **EstCon**. The call_generator received a wake-up signal from the call_scheduler and the signal contains the response for the connection request sent to the AAL module. If the signal is an established connection signal, this state generates a delayed self interrupt to signal the call termination and then generates an immediate interrupt to signal generation of the first data packet. Control immediately transfers to the **data gen** state. If the signal from the call_scheduler indicates call rejection by the AAL module, this state transfers control to the **reschedule** state.

- **data gen**. This state continues to generate data packets until the call completes or the call_scheduler sends a network release indication. Upon receiving the interrupt signaling call completion, this state sends a call release request to the AAL module and transitions to the **RelCon** state. If the network signals with a connection release, this state transitions to the **reschedule** state.

- **RelCon**. After completing the call and sending the release request, this state sleeps until the network responds with a release confirmed signal. This signal forces the transition to the **call done** state.

- **reschedule**. When a call is rejected by the network, or the network signals a call release during data transmission, the current call requires rescheduling. This state creates a call request that is forwarded to the call_scheduler module. This state then signals the call_scheduler module that the channel is available and then destroys this process.

- **call done**. When the call is completed and all network resource are released, this state signals the call_scheduler module that the channel is available and then destroys this process.

E. SUMMARY

In this chapter we have explained, in detail, the design and implementation of our OPNET Model of the BADD network. Due to the complexity of the simulation model, we have only been able to elaborate on the most significant aspects of our design. Additional information can be found in Appendices C through F which contain the OPNET reports for the processes designed to support our simulations.

Now that our simulation model is defined it is time to put it to work. The next chapter describes our simulation testing plan, our results, and conclusions.

VII. ANALYSIS AND RECOMMENDATIONS

In this chapter we describe our experiments and the results of our testing. Since most of our efforts concentrated on the construction of the simulation of BADD, we provide a section on our observations from using OPNET as a network development tool as well as one on lessons learned. We also make recommendations for new features that we would like to see in a tool for building prototype networks. Finally, we describe possible areas of future research that build upon our simulation model.

A. TESTING

We conducted multiple runs of our simulator using both the default FIFO scheduler and our Greedy scheduler implementation. We initially validated our basic model using only a single data generation source and channel. We validated our design and ensured the correctness of the values generated by the simulation. After verifying the correctness of our basic model, we ran additional simulations varying the parameters described in Table XII using small data sets as reflected by the short durations of the calls and the small numbers of channels and requesters. Small data sets were used to determine the sensitivity of our simulations to changes of various parameters. Next we conducted additional runs to determine whether our analysis of the smaller data sets holds for the larger, more realistic data sets. Finally, we used an exponential distribution for the interarrival rates and wait times of the call requesters to determine whether our analysis held for more general cases.

1. Simulation Duration

We varied the runtime of our simulation between 10 and 50 seconds. A runtime of at least 10 seconds was needed to allow the simulation to complete its warmup phase, a phase that is common to all simulations [Ref. 17]. We selected the maximum of 50 seconds because, by that time, the bit throughput clearly approached its asymptotic limit as reflected in Figure 52. OPNET simulations complete in any of three ways:

Parameter	Range
Duration of Simulation	10 to 50 seconds
Number of Requesters	1 to 16
Number of Channels	2 to 20
Data Sources and Channel Characterization	nearly homogeneous to heterogeneous

Table XII. Simulation Parameters and Ranges of Values

(i) when the simulated time reaches a certain value, (ii) when a module within the network reaches a final state, or (iii) in unusual circumstances, when the event list is empty. We used the first option and set a simulation completion time to end our simulation.

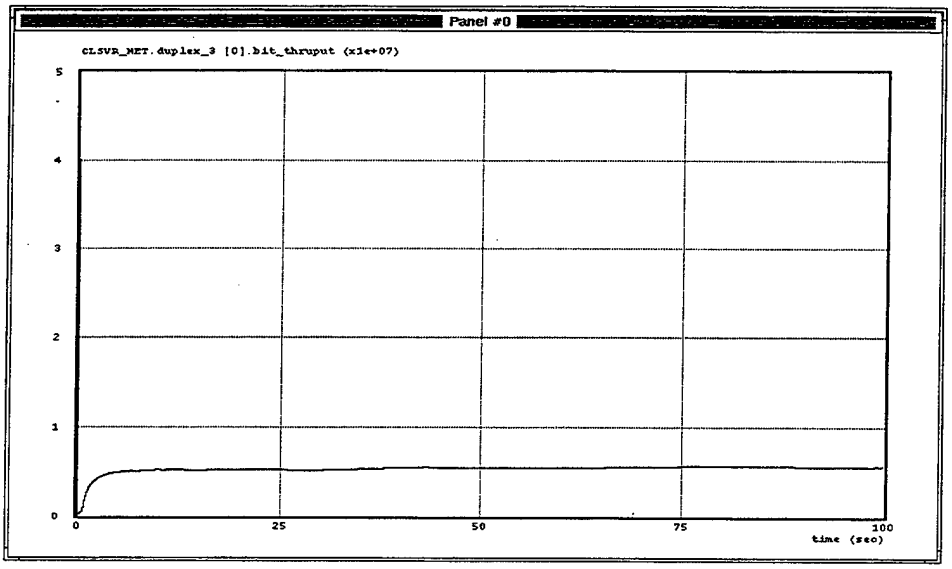


Figure 52. Greedy bit throughput results after 100 Seconds.

2. Channels

We varied the number of channels between 2 and 20 in order to provide a wide range for analysis. The BADD architecture proposes to use 1024 channels on this 8 Megabit link. This configuration requires that the bandwidth on individual channels be

very small. Since one of the requirements of BADD is to broadcast bandwidth intensive application we believe that some large channels are required. The use of a few large bandwidth channels brings down the total number of usable channels drastically. In fact, we constructed a bandwidth allocation Lotus 1-2-3 spreadsheet and determined that we could get a good mix of channels, of standard size, if we limit the total number of channels to 20. Typically, telecommunication channels are allocated in bits-per-second (bps) with standard capacities being 64 kilobits-per-second (Kbps), 128 Kbps, 256 Kbps, 512 Kbps, and 1.544 megabits-per-second (Mbps). We confined ourselves to using a mix of these standard channel capacities as shown in Table XIII. We used the smaller values in the table to perform our sensitivity analysis on our other test parameters. We recognized that BADD's 8Mbps channel would not be completely utilized during the simulations with 8 or fewer channels. Nonetheless, these experiments permitted us to perform the sensitivity analysis to determine which parameters most influenced the simulation results.

In Table XIII we list 2 columns, heterogeneous and homogeneous channel mixes. We provide these columns to support our definition of heterogeneous and homogeneous experiments. For heterogeneous experiments we will use heterogeneous data sources over heterogeneous channels. Similarly, for homogeneous experiments we will use homogeneous data sources over homogeneous channels. These data source classifications will be discussed fully in the next section.

3. Number of Call Requesters and Data Sources

We varied the number of requesters between 2 and 16. We were constrained to 16 by OPNET's limitations on the numbers of requesters that can be connected into the AAL module. The OPNET GUI for building nodes does not allow the lines that represent data streams that connect modules to intersect. Because of the physical limits of connecting data streams graphically, we experienced difficulty when we tried to use more than 16 requesters.

Table XIV lists the variety of data sources that we used for testing with smaller

Number of Channels Allocated	Heterogeneous Mix of Channels	Homogeneous Mix of Channels
2	1.544 Mbps (x1) 256 Kbps (x1)	512 Kbps (x2)
4	1.544 Mbps (x1) 512 Kbps (x1) 256 Kbps (x1)	512 Kbps (x2) 256 Kbps (x2)
8	1.544 Mbps (x2) 512 Kbps (x2) 256 Kbps (x2) 128 Kbps (x1) 64 Kbps (x1)	512 Kbps (x4) 256 Kbps (x4)
20	1.544 Mbps (x3) 512 Kbps (x2) 256 Kbps (x4) 128 Kbps (x4) 64 Kbps (x7)	512 Kbps (x6) 256 Kbps (x8) 128 Kbps (x6)

Table XIII. BADD Channel Allocations for Various Channel Configurations.

data sets. Table XV shows the various mixes of application types that we used in these simulations. In our smaller tests, as well as with our larger tests, we wanted a mix of applications that would be representative of those that we would expect to see sent over the BADD network. The values in the table were loosely based upon those used in Joint Task Force (JTF) Advanced Technology Demonstration (ATD) experiment in 1995 [Ref. 19]. After completing simulations on the smaller data sets, we used exactly the same data as that used in the JTF ATD experiment as shown in Table XVI. Table XVII shows the mix of requesters we used for our heterogeneous and homogeneous runs. Below we expand upon the sizes of messages generated by the various JTF ATD applications.

1. Database Views (DB) - The mean of message sizes is 1 Mbyte. We used both this value and smaller value of 500 Kbytes with a constant distribution.
2. Email - Ranged from 100 bytes for text only systems to 1 Mbyte systems capable of handling graphical attachments. We selected values of 1 Mbyte, 250

Kbytes, and 500 bytes to represent these ranges using a constant distribution.

3. Web Masters - Ranged from 500 bytes to 50 Kbytes. We selected values of 1 Kbyte and 50 Kbytes to represent this range. Again we used a constant distribution to generate these packets in our simulation.

4. Video - This value did not come from the JTF ATD experiment. We calculated this value based upon a UAV transmission rate of 64Kbps. We assumed that a commander would be interested in the UAV's flight over his area of interest and that the time required for fly over would be 2 minutes. Based upon this time constraint, and the data rate the UAV sends data to the ground station, we calculated the size of 1 MByte for this message (message size = 64Kbps * 60 seconds * 2 minutes * $(\frac{1byte}{8bits})$).

Data Source	Wait	Call Size	BPS Production Rate
Large DB View	1.0	338 Kbytes	1.35M
Video	0.2	135 Kbytes	1.0M
Large Email	0.1	63 Kbytes	507K
Large Web	0.5	32 Kbytes	256K
Small DB View	0.001	206 bytes	165K
Medium Email	0.01	16 Kbytes	126K
Small Web	0.01	9.9 Kbytes	79K
Small Email	0.0011	675 bytes	54K

Table XIV. Representative Simulation Data Sources

We now discuss our choice of homogeneous and heterogeneous mix of data sources in more detail. We use the term homogeneous to mean that calls generated by the data sources have similar bit production rates. A heterogeneous mix means that there is more variety in these production rates. An example of a heterogeneous set of sources that we used had one 1.581 Mbps source, one 501.894 Kbps source, one 250.95 Kbps source. and one 65.34 Kbps source. A homogeneous set of 4 sources consisted of two 501.894 Kbps sources and two 250.95 Kbps sources.

Number of Requesters Allocated	Heterogeneous Mix of Requesters	Homogeneous Mix of Requesters
2	Large DB Views (x1) Small Email (x1)	Large Email (x2)
4	Large DB View (x1) Large Email (x1) Small DB Views (x1) Small Email (x1)	Large Email (x2) Large Web (x2)
8	Large DB View (x1) Video (x1) Large Email (x1) Large Web (x1) Small DB View (x1) Medium Email (x1) Small Web (x1) Small Email (x1)	Large Email (x4) Large Web (x4)

Table XV. Requester Configurations for Testing Small Data Sets

B. RESULTS

In Figure 53 we provide the results of our testing for both of the schedulers using small data sets. The column label indicates the runtimes of our simulations in seconds. The row labels indicate the configuration of the requesters and the number of servicing channels. The rows are each divided into sub-rows. The top entry for each

Data Source	Wait	Call Size	BPS Production Rate
Large DB Views	0.0007	1.0 Mbytes	1.581M
Large Email	0.0022	1.0 Mbytes	501.894K
2 Min Video	0.0007	1.0 Mbytes	1.581M
Small DB Views	0.0022	500 Kbytes	501.894K
Medium Email	0.0044	250 Kbytes	250.95K
Large Web View	0.009	50 Kbytes	122.69K
Small Web View	0.0169	1 Kbytes	65.34K
Small Email	0.0169	500 bytes	65.34K

Table XVI. Realistic Simulation Data Sources (Derived From [Ref. 19]).

Number of Requesters Allocated	Heterogeneous Mix of Requesters	Homogeneous Mix of Requesters
16	Large DB Views (x2) Large Email (x2) 2 Min Video (x2) Small DB Views (x2) Medium Email (x2) Large Web View (x2) Small Web View (x2) Small Email (x2)	Small DB Views (x4) Medium Email (x8) Large Web View (x4)

Table XVII. Channel Allocations for 16 Requester Configurations.

sub-row is the average bit throughput. The bottom entry is the completion time of the last scheduled transmission if all transmissions are allowed to run to completion. Note that we also simultaneously vary the mix of the data requesters and channels as depicted by the partitioning of the columns.

We provide the results of our testing for the realistically sized data in Figure 54 in the same format as Figure 53. We eliminated the 10 and 25 second runtimes because they do not provide any new data. (Instead, we looked at varying the input from the requesters from a constant distribution to an exponential distribution.) In the next section we provide the analysis of these results.

C. ANALYSIS OF RESULTS

This section provides the analysis of testing that allowed us to make certain conclusions about the nature of the BADD architecture with different schedulers.

1. Effects of Satellite Delay on ATM Protocol

OPNET provides direct support for modeling duplex ATM networks. Earlier we described how we modified these models to approximate BADD's simplex ATM protocol. These models allow the user to insert delays, which are normally small, between ATM switches. However, since the BADD architecture uses a Global Broad-

GREEDY

Run Duration (seconds)

Sources/ Channels	Heterogeneous Requesters			Homogeneous Requesters		
	10	25	50	10	25	50
2/4	0.8M	0.88M	0.9M	0.82M	0.86M	0.87M
	10.96	26.65	54.3	11.6	29.1	57.85
4/2	0.32M	0.35M	0.38M	0.67M	0.69M	0.69M
	44.06	109.87	221.5	20.35	50.35	99.11
4/4	0.7M	0.76M	0.79M	1.25M	1.28M	1.3M
	22.25	55.31	111.32	11.6	29.1	57.85
4/8	1.4M	1.52M	1.57M	1.5M	1.56M	1.61M
	14.46	28.51	56.65	11.0	25.8	51.25
8/4	0.8M	0.87M	0.9M	1.25M	1.27M	1.28M
	34.15	83.41	165.87	22.85	57.85	115.36
8/8	1.8M	1.8M	1.8M	2.5M	2.51M	2.52M
	17.90	42.21	83.26	12.25	29.75	58.5

FIFO

Run Duration (seconds)

Heterogeneous Requesters			Homogeneous Requesters		
10	25	50	10	25	50
0.72M	0.74M	0.77M	0.84M	0.86M	0.87M
12.95	32.25	60.16	11.6	29.1	57.85
0.4M	0.4M	0.38M	0.71M	0.7M	0.7M
45.6	117.47	237.2	20.35	50.35	99.11
0.7M	0.7M	0.68M	1.10M	1.23M	1.27M
27.95	67.56	136.92	12.85	30.35	59.10
1.3M	1.35M	1.4M	1.56M	1.74M	1.76M
14.65	33.76	69.5	11.0	25.8	51.1
0.7M	0.72M	0.75M	1.2M	1.26M	1.28M
45.71	107.86	214.97	24.1	59.1	116.6
1.3M	1.48M	1.52M	2.25M	2.46M	2.52M
23.95	56.76	117.16	13.5	31.0	60.35

Figure 53. Representative Test Results

	GREEDY		FIFO	
	Run Duration (seconds)		Run Duration (seconds)	
	50	50	50	50
Constant	5.4M	4.4M	3.7M	3.95M
	92.79	63.25	174.42	67.28
Exponential	4.8M	3.1M	4.0M	2.74M
	81.73	113.34	140.47	101.34
	Heterogeneous Requesters	Homogeneous Requesters	Heterogeneous Requesters	Homogeneous Requesters

Figure 54. Realistic Test Results

cast Satellite (GBS) satellite in geosynchronous orbit, we must use a fairly large delay of 0.250 seconds, which is more typical of geosynchronous satellite propagation [Ref. 6].

For comparison, we did execute the original OPNET dynamic virtual channel duplex protocol using the 0.25 second delay. Our first observation is that the duplex ATM protocol is not efficient when using links with large delays because of the setup signaling requirements in the protocol. Figure 55 shows that the these requirements require the channel to be idle for large periods of time while awaiting responses to establish and terminate call setups. This idle time clearly reduces the amount of data transmitted over the network and lowers the bit throughput rate. Therefore we conclude that although the BADD designers chose a simplex implementation for a different reason, a duplex implementation for a BADD-like architectures should never be considered due to the low bit throughput rate caused by the combination of large propagation delays with dynamic virtual channels.

We see two areas that are worthy of further investigation. One is the incorporation into OPNET of a simplex ATM protocol. The second is an investigation of

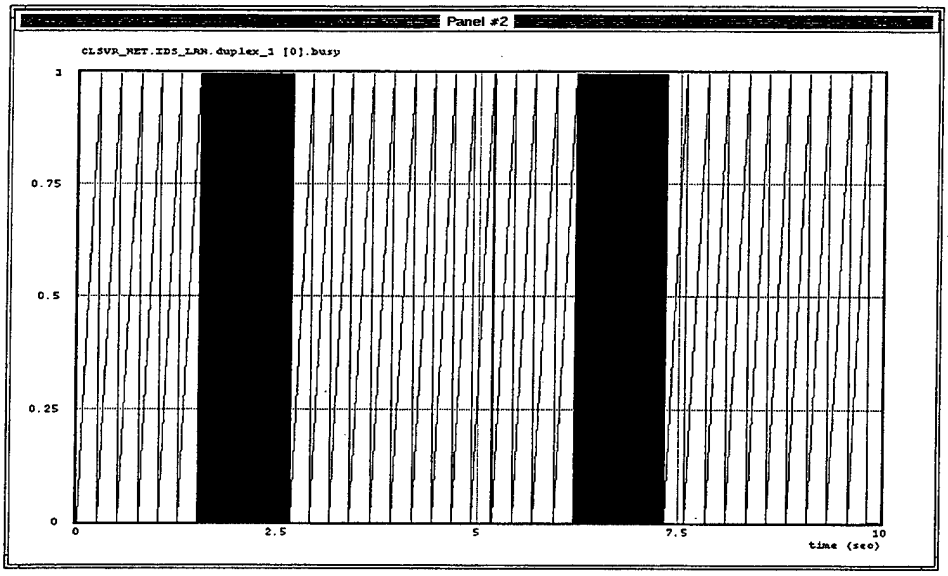


Figure 55. Effects of Large Delay on Link Utilization

the use of Low Earth Orbit (LEO) satellites rather than geosynchronous satellites in BADD. The use of LEO satellites would require modification of the protocol because the LEO satellite is moving; however, the propagation delays would be significantly decreased to approximately .005 seconds, a value from the planned Teledesic Satellite System orbiting between 695 and 705 kilometers [Ref. 20].

2. Effects of Varying Simulation Parameters

Our experimental results shown above indicate that scheduler performance was independent of

1. the number of sources,
2. the number of calls, and
3. the duration of our simulation.

However it depends heavily on the heterogeneity of both the sources and the channels. We found that the FIFO algorithm occasionally outperforms the Greedy algorithm with simulations run with homogeneous sources and channels, that is when the bandwidth requirements of the sources were nearly identical to each other as well

as the capability of the channels. However, in all other cases the Greedy scheduler provides a better schedule. In the following subsections, we provide results of simulations with the 16 requesters and the 20 channels defined in the previous section.

a. Completion Time Observations

Using the multiple sources shown in Table XVII, we found that the total channel utilization using either the Greedy or FIFO scheduler is approximately the same. Figure 56 compares the completion times of channels using both the Greedy and FIFO schedulers. While the average channel completion time using either of these schedulers is approximately 72 seconds, the Greedy scheduler provides a much more uniform set of completion times. When evaluating the worst completion times of the schedules, we find that Greedy's worst completion time is 92.79 seconds whereas FIFO's worst time is 174.42 seconds. The worst FIFO channel takes 88 percent longer than the worst Greedy channel.

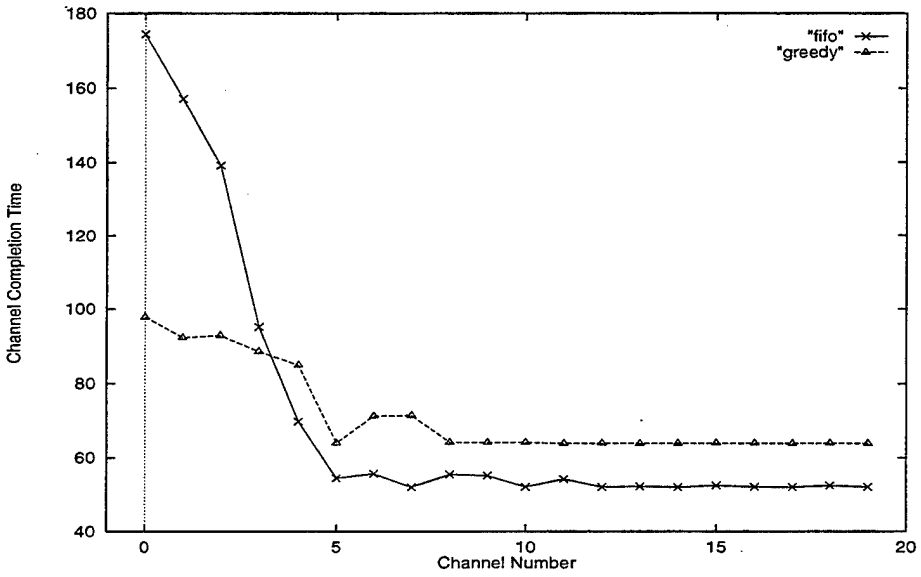


Figure 56. Greedy vs. FIFO Channel Completion Times for Heterogeneous Sources and Channels

To help explain these results, Figure 57 shows the Estimated Time of Completion (ETC) Matrix that provides the key reason why both schedulers have the same average completion time. When our requester generates a call, the first step is

	VC1	VC2	VC3
CALL 1	100	100	
CALL 2	10	10	10
CALL 3	35		

Figure 57. Estimated Time of Completion Matrix

to determine which channels can support the transmission. In our ETC matrix we display some empty entries. These empty entries are result of the channels inability to support the traffic and need not be evaluated by the scheduler. We also notice that each call requires the same amount of time on each virtual channel. The reason why all VCs have the same value for a given call is because we considered only one signaling characteristic, transmission rate. When other characteristics such as bit error rate are taken into consideration we would not see the same values in all supporting VCs for a particular call. Our implementation, with the same estimated completion values for each call, independent of VC, causes both schedulers to have the same finishing times because the number of calls requested are the same and the servicing times for the calls are the same. In other words, even though the schedulers place the calls on different channels, the aggregate time to run the calls will be the same for both schedulers.

b. Bit Throughput Observations

Using a heterogeneous set of sources we found that the use of the Greedy algorithm yields better bit throughput than the use of the FIFO algorithm. Figures 58 and 59 show that the bit throughput for Greedy averages approximately 5.4 Mbps

while FIFO's bit throughput averages approximately 3.7 Mbps. These results show a 46 percent greater average bit throughput for a representative set of sources and channels.

c. Calls Scheduled vs Calls Completed

Figures 60 and 61 show a distribution diagram of the number of calls scheduled (dashed lines) vs. the number of calls completed on each of the channels (solid lines). We see that the number of calls scheduled for each channel is much more uniform for the FIFO Algorithm. This demonstrates an innate failure of the FIFO algorithm in that it assigns calls to channels based only upon the number of calls scheduled. FIFO does not consider the length of the calls, but only that the number of calls are evenly split between channels. We also observe that the average difference in the number of calls scheduled versus the number of calls completed is approximately the same for both algorithms, approximately 42 for our representative run. The difference between the algorithms is that the variance of this difference is much higher for the Greedy algorithm. This observation makes sense because the Greedy algorithm does not attempt to balance the number of calls, but rather attempts to minimize the completion times.

3. OPNET Observations and Recommendations

OPNET is an extremely powerful network simulation tool. We found their libraries useful in modifying our network and are grateful that all of their source code is provided. We found that once our basic model was developed, that it was very easy to modify with the GUI. We also appreciated that a discrete event simulator, supporting different distributions, was already present as well as an analysis tool that graphed output with little effort from the user. Clearly OPNET is a quality networking tool. However, there are challenges in using state-of-the-art products to model next generation networks. In this section we will highlight some of the lessons we learned and provide recommendations for OPNET and other modeling tools.

The approach taken in BADD is to keep things simple at first and then incre-

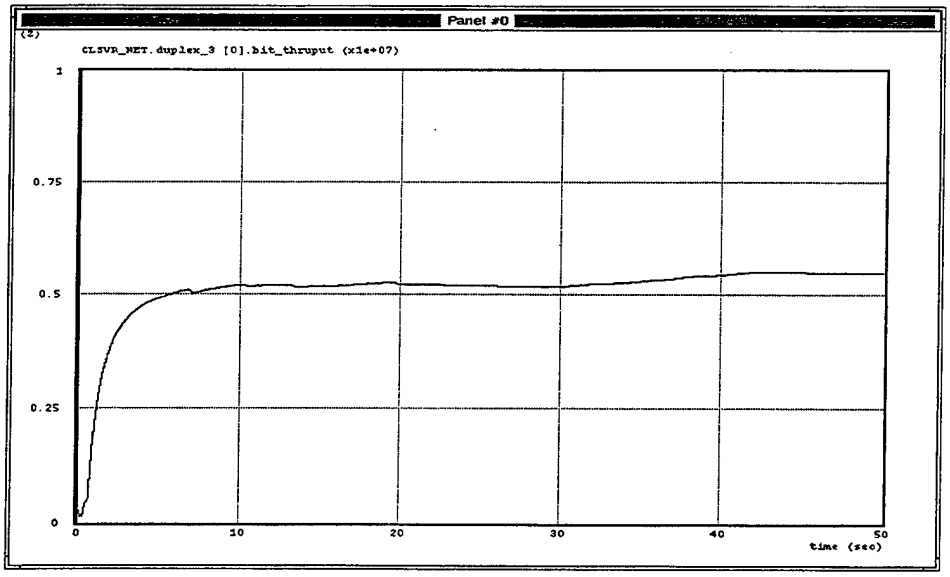


Figure 58. Greedy Bit Throughput for Heterogeneous Sources and Channels

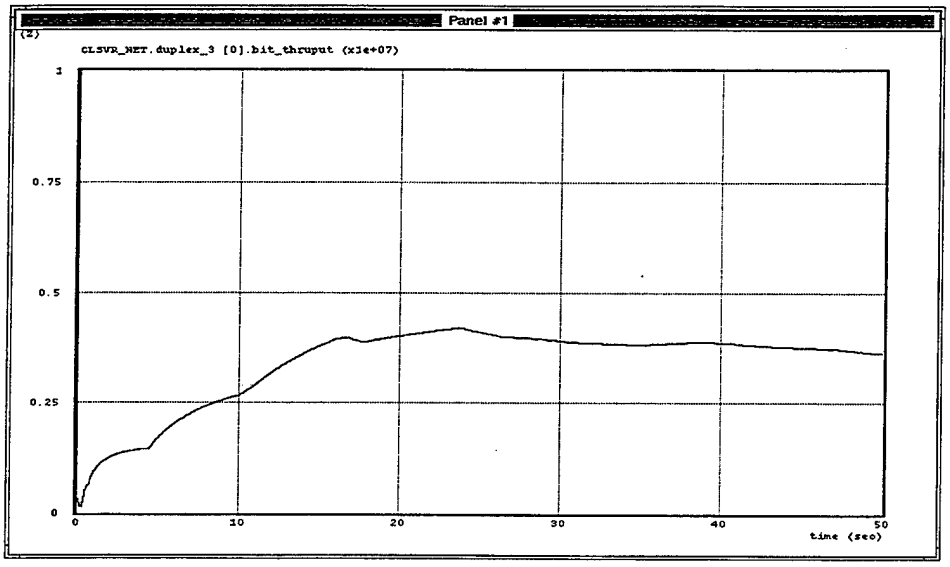


Figure 59. FIFO Bit Throughput for Heterogeneous Sources and Channels

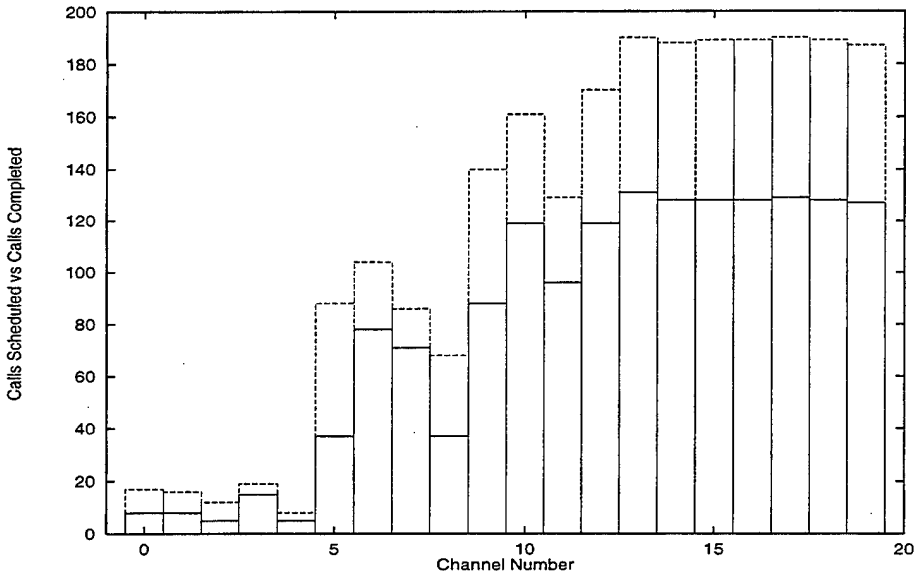


Figure 60. Greedy Algorithm: Calls Scheduled vs. Calls Completed for Heterogeneous Sources and Channels

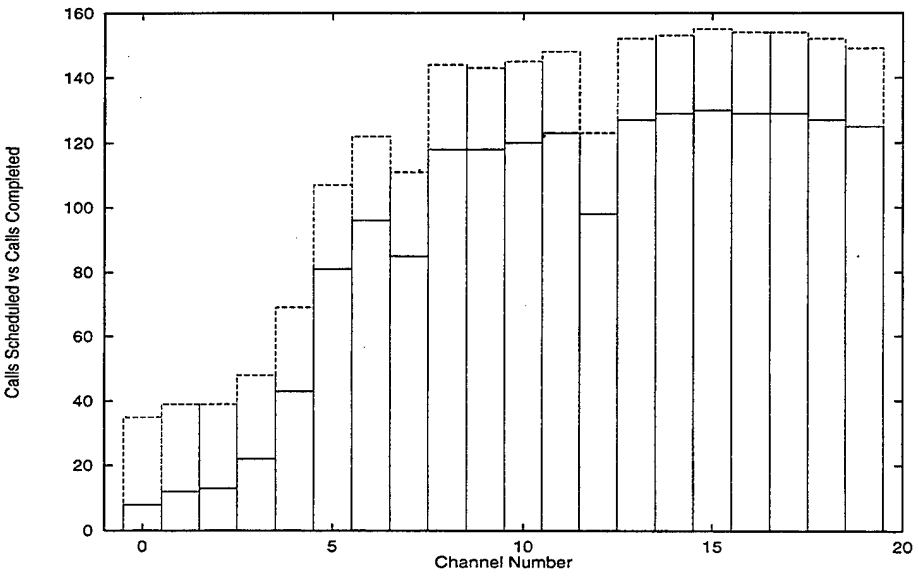


Figure 61. FIFO Algorithm: Calls Scheduled vs. Calls Completed for Heterogeneous Sources and Channels

mentally improve the features of the architecture as the users become more comfortable with the equipment and as industry establishes more robust capabilities in the commercial market. Therefore, the BADD network simplifies the ATM architecture by stripping out much of the functionality of ATM. OPNET's models are based on using most of the functionalities defined in the standard duplex ATM protocol. Because of BADD's design decision to use static virtual channels on simplex links we were forced to greatly modify the OPNET code and support utilities.

At first we found it difficult to design our network using their standard models because they differed substantially from BADD. The tutorials had given us a false sense of security and we began our design with the intent of making modest changes to the duplex implementation. As we became more familiar with the tool, we recognized that we would have to rewrite the OPNET code for setup and tear down of dynamic virtual paths and virtual channels. Below we enumerate some of the lessons learned from our simulation design, provide our observations for future users, and offer suggestions for additional features we would like to see OPNET provide in the future.

OPNET requires the commitment of a large amount of time before the user will feel comfortable using the tools for advanced research. With no training, a vast amount of our design time was spent developing a basic working network model. In order to gain maximum benefit from this simulation tool, the user should have a thorough understanding of the network and the associated protocols used within it. We recommend the user read the following portions of the OPNET manuals in the order we specify.

1. Complete the Introduction to OPNET Tutorial, Basic Model Tutorial and one other tutorial to get a feel for OPNET.
2. Read the first Modeling Manual in detail on the Modeling Overview, Modeling Framework, and Process Domain Definitions.
3. Read the particular section on the model or protocol that the user is interested in modeling.

If we had carefully read these sections in the above order, we would have saved a month of our time learning the tool. We continually found ourselves going back to these sections to refresh our understanding of how OPNET worked.

Below we make some suggestions for enhancing OPNET to make it more useful for tomorrow's network designers. We separate our suggestions into major and minor categories.

a. Major Enhancements Recommended for OPNET

1. MIL 3 should consider providing packet generators and queue models at the process and node level. Currently, OPNET implements numerous functions, such as packet generators and queues, only at the node level. During our design, we required these functions at the process level which forced us to write these functions as processes within our model.

2. We believe that simplex ATM, particularly when used in conjunction with geosynchronous satellite broadcast, will be an important protocol of the future. We therefore believe that MIL 3 should consider providing a simplex ATM protocol.

3. Running OPNET simulations from the command line is neither intuitive nor user friendly. During our model development, our system's administrator upgraded the operating system and we lost access to the OPNET Graphical User Interface for three weeks. The External Interfaces Manual describes only basic command line syntax and the command line options are spread over several different chapters in the manual. Giving the command line syntax with all available options in one section would drastically improve the usefulness of the manual.

4. MIL 3 should consider providing a GUI utility to allow the user to save simulation runs, enabled with traces, to a file for further analysis. We needed to run self-produced scripts from the command line in order to save data into scripts for analysis. Some operations required invocation of debug operations in order to capture the status of data throughout the simulation.

b. Minor Enhancements Recommended for OPNET

1. We recommend that MIL 3 look at reimplementing their code to calculate the Peak Cell Rate (PCR) in the ams_traf_gen process as floating point division. The current implementation does integer division. Subsequently a conditional checks the value of the PCR and assigns the value of 1 to all calculations less than one.

2. We were constrained by OPNET's limitations on the numbers of requesters that can be fed into the AAL module. OPNET should provide a method, at the module level, that allows the modeler to include large numbers of data generators into the AAL module.

4. Parallelization of Scheduling Algorithms

We examined the possibility of using parallel processors to improve the speedup of our scheduling algorithms. These efforts were done independent of our OPNET simulation.

We used a Silicon Graphics Challenge, with four processors to parallelize our Greedy code. Table XVIII shows the time in seconds to complete scheduling based on the number of calls. The speedup calculations are based upon the runtime of the C code on one processor against the runtime of the C code on all 4 processors.

Calls	Channels	Seq(C)	Para(C)	Speedup
20	16	0.104	0.542	0.19
100	16	0.12	0.604	0.20
1000	16	2.967	4.318	0.69
10000	16	287.053	364.829	0.79
20000	16	1447.164	1567.002	0.92
30000	16	2587.132	2314.182	1.12
40000	16	4584.385	3738.235	1.23

Table XVIII. SGI Speedup Calculations
[SGI Speedup Calculations, all times are given in microseconds.]

D. RECOMMENDATIONS FOR FUTURE RESEARCH

1. Optimal Scheduling Times

OPNET discrete event simulations service events under the assumption that they take no time to execute. This is necessary to support the overhead of running a simulation. A side effect of this design is that calls to our scheduling algorithms do not require any simulation time. This is clearly not realistic. We investigated

incorporating the run times into our schedulers, independent of the simulation, to gain some understanding of how much time it takes to run the scheduling algorithms. We found that the runtime of the algorithm was proportional to size of squaring our input to the algorithm. We used this rough estimate to calculate a delay for scheduling but did not include this modification in the simulation due to time constraints. The inclusion of this modification is absolutely necessary to perform a cost-benefit analysis of integrating the scheduling. It is also necessary in determining the optimal time to wait before calling the scheduler.

2. Implementation with Static Virtual Paths vs. Static Virtual Channels

We recommend that simulation of the BADD network be modeled with static virtual paths rather than static virtual channels. This model would be more natural to the OPNET ATM protocols and would preserve the flexibility of dynamic virtual channel allocation within the static virtual paths. The preservation of dynamic allocation would provide flexibility for schedules to use the large bandwidth paths to send multiple small calls when they are not sending large calls over the path.

3. Inclusion of Attributes in Scheduling Algorithm

Our current scheduler makes its decisions based upon completion time only. A better implementation would include considerations for other attributes such as bit error rate and jitter characteristics of the various channels. A more comprehensive scheduler could accomplish this by applying weighting factors for each desired attribute.

E. SUMMARY

The use of OPNET to model communication networks is highly advantageous because it provides an extensive array of tools to support simulation development and analysis. When developing new protocols there is significant time required to understand the complexities of OPNET before the user can achieve their desired

modeling results.

The use of the $O(n^2m)$ Greedy algorithm will decrease the time required to completely process all calls. Our results indicate that the BADD program should consider integrating this algorithm into their current architecture in order to increase the performance of the network.

APPENDIX A. ABBREVIATIONS AND DEFINITIONS

The following listing of abbreviations and definitions includes an abridged and extended version of Cisco Internetworking Terms and Acronyms. The abridgement is based on terms used in this thesis. The original glossary can be found at http://www.erl.noaa.gov/noc/cisco/data/doc/cintrnet/ita.htm

- **ACK Flag.** A bit that, when set, alerts the protocol that the acknowledgment number is valid.
- **ACTD** Advanced Concepts Technology Demonstration.
- **AHPCA** Advanced High Performance Computing Applications.
- **Address Translation.** The process of converting external addresses into standardized network addresses and vice versa. It facilitates the interconnection of multiple networks in which each have their own addressing scheme.
- **Analog** A type of transmission in which a continuously variable signal encodes an infinite number of values for the information being sent. (Compare with "digital.")
- **ANSI** The American National Standards Institute is a US-based organization that develops standards and defines interfaces for telecommunications systems.
- **ABCS** Army Battle Command System.
- **Asynchronous** A term used to describe any transmission technique that does not require a common clock between the two communicating devices, but instead derives timing signals from special bits or characters (i.e., start/stop bits, flag characters) in the data stream itself. (Compare with "synchronous.")
- **Asynchronous Transfer Mode** (ATM) A form of digital transmission based on the transfer of units of information known as cells. It is suitable for the transmission of image, voice, video, and data.
- **ATM** Asynchronous Transfer Mode.

- **ATM Adaptation Layer** (AAL) The AAL translates digital voice, image, video, and data signals into the ATM cell format and vice versa. Five AALs are defined:
  - **AAL1** supports connection-oriented services needing constant bit rates and specific timing and delay requirements. (e.g., DS-3 circuit)
  - **AAL2** supports connection-oriented services needing variable bit rates. (e.g., certain video transmission schemes)
  - **AAL3/4** supports both connectionless and connection-oriented variable rate services.
  - **AAL5** supports connection-oriented variable bit rate data services. AKA: Simple and Efficient Adaptation Layer (SEAL)
- **ATM Application Program Interface** (API) While no standard ATM API exists, the ATM Forum is developing an API that enables application developers to use ATM's quality of service and traffic management features. In the meantime, FORE Systems is one the vendors that supply an API library with its line of ATM network adapter cards. Services such as guaranteed bandwidth reservation, per connection selection of AAL5 or AAL3/4, and multicasting with dynamic addition and deletion of recipients are made available to the application. Applications that use FORE's ATM API bypass inefficiencies and overhead associated with typical TCP/IP protocol stacks.
- **ATM Layer** The protocol layer that relays cells from one ATM node to another. It handles most of the processing and routing activities including: each cell's ATM header, cell muxing/demuxing, header validation, payload-type identification, quality-of-service specification, prioritization, and flow control.
- **Available Bit Rate** (ABR) A type of traffic for which the ATM network attempts to meet that traffic's bandwidth requirements. It does not guarantee a specific amount of bandwidth and the end station must retransmit any information that did not reach the far end.
- **BADD** Battlefield Awarness and Data Dissemination.
- **BC2A** Bosnia Command and Control Augmentation Initiative.
- **Bandwidth** A measure of capacity, usually, the capacity of a communications line to transmit voice, data, video, or image traffic through a network. Bandwidth is usually expressed in bits per second (bps), thousands of bits per second (kbps), millions of bits per second (Mbps), or billions of bits per second (Gbps).

- **Border Gateway Protocol** (BGP) A protocol used by gateway devices (e.g., routers) in an internetwork, connecting autonomous networks. Based on the exterior gateway protocol.

- **Broadband** A service or system requiring transmission channels capable of supporting rates greater than the Integrated Services Digital Network (ISDN) primary rate (1.544 Mbps).

- **Broadcast (Messages)** Transmissions sent to all stations (or nodes, or devices) attached to the network.

- **BMC** Broadcast Management Center.

- **Broadcast and Unknown Server** (BUS) In LAN Emulation, a LAN Emulation server provides address resolution between LAN addresses and ATM network addresses. Multicast traffic and unknown traffic (i.e., any data for which the sender has not obtained an ATM address) require different procedures and are handled by the broadcast and unknown server.

- **Buffer** An area of storage that provides an uninterrupted flow of data between two computing devices.

- **CCITT** The Consultative Committee on International Telephony and Telegraphy, now the International Telecommunications Union (ITU), is an international organization that develops standards and defines interfaces for telecommunications systems.

- **CNN** Cable News Network.

- **Cell** A transmission unit of fixed length used in cell relay transmission techniques such as ATM. An ATM cell is made up of 53 bytes (octets) including a 5-byte header and a 48-byte data payload.

- **Cell Delay Variation** (CVD) A measurement of the allowable variation in delay between the reception of one cell and the next. (Usually expressed in thousandths of a second, or milliseconds (msec.). Important in the transmission of voice and video traffic, CDV measurements determine whether or not cells are arriving at the far end too late to reconstruct a valid packet.

- **Cell Relay** Any transmission technique that uses packets of a fixed length. ATM, for example, is a version of cell relay using 53-byte cells. Other versions use cells of a different length.

- **Cell Loss Priority** (CLP) A bit in the ATM cell header that indicates the cell's transmission priority. If the bit is set (value=1), the cell is eligible to be discarded, if it is not (value=0), it cannot be discarded.

- **Circuit Switching** A switching technique in which a dedicated path is set up between the transmitting device and the receiving device, remaining in place for the duration of the connection. (e.g., a plain old telephone call is a circuit-switched connection)

- **Common Part Convergence Sublayer** (CPCS) The part of an ATM adaptation layer's convergence sublayer that does not vary with the type of traffic being sent.

- **Connection Admission Control** (CAC) Methods used to control the set up of switched virtual circuits. For example, one method is known as overbooking and assumes that active connections are not using all of the available bandwidth. Another method, known as full booking, limits network access once all bandwidth is committed. Only connections that request acceptable traffic parameters are permitted.

- **Connection-oriented** A type of communication in which an assigned path must exist between a sender and a receiver before a transmission occurs. (e.g., circuit switching) ATM networks are connection-oriented.

- **Connectionless** A type of communication in which no fixed path exists between a sender and receiver, even during a transmission. (e.g., packet switching) Shared media LANs are connectionless.

- **COTS** commercial off-the-shelf.

- **Constant Bit Rate** (CBR) A type of traffic that requires a continuous, specific amount of bandwidth over the ATM network (e.g., digital information such as video and digitized voice).

- **C4I** Command, Control, Computer, and Intelligence.

- **Digital** A type of transmission that encodes a discrete value (e.g., "0" or "1") for each unit of information being encoded. (Compare with "analog.")

- **DARPA** Defense Advanced Research and Project Agency.

- **DIA** Defense Intelligence Agency.

- **DIM** Downlink Information Manager.

- **DISN LES** Defense Information Systems Network Leading Edge Services.

- **DMA** Defense Mapping Agency.

- **DoD** Department of Defense.

- **Dual Leaky Buckets** A method employed by ATM switches to perform flow control (traffic policing) on ATM network connections. Hardware in the switch monitors each connection to determine whether it has exceeded its negotiated rate. Dual leaky buckets refers to the use of one "bucket" to monitor the average or sustained rate and one to monitor the peak rate.
- **EPLRS VHSIC** Enhanced Position Location Reporting System Very High Speed Integrated Circuit.
- **FIFO** First In First Out.
- **FIN Flag**. This bit is set when either the sender or receiver finishes with a connection.
- **Frame** Variable length packet of data used by traditional LANs such as Ethernet and Token Ring as well as WAN services such as X.25 or Frame Relay. An edge switch will take frames and divide them into fixed-length cells using an AAL format. A destination edge switch will take the cells and reconstitute them into frames for final delivery.
- **GBS** Global Broadcast System.
- **Generic Flow Control** (GFC) The first four bits of the ATM UNI cell header; used when passing cells to/from the UNI. Setting any of the bits in this field indicates to the end station that the ATM switch can implement some form of congestion control.
- **Gigabits Per Second** (Gbps) A digital transmission speed of billions of bits per second.
- **GUI** Graphical User Interface.
- **Header Error Control** (HEC) An 8-bit Cyclic Redundancy Code (CRC) computed on all fields in an ATM header; capable of detecting single bit and certain multiple bit errors. HEC is used by the Physical Layer for cell delineation.
- **HMMWV** High Mobility Military Wheeled Vehicle,
- **ICI** Interface Control Information. Used by OPNET processes.
- **IDS** Information Dissemination Server.
- **IMETS** Integrated Meteorological System.
- **IRD** Integrated Receiver Decoders.

- **ISO** International Standards Organization.
- **IST** Integrated Systems Technology.
- **ITU-T** International Telecommunication Union Telecommunication Standardization Sector.
- **Internet Protocol (IP) Address** An identifier for a network node; expressed as four fields separated by decimal points (e.g., 136.19.0.5.); IP address is site-dependent and assigned by a network administrator.
- **IP-over-ATM** The adaptation of TCP/IP and its address resolution protocol for transmission over an ATM network. It is defined by the IETF in RFCs 1483 and 1577. It puts IP packets and ARP requests directly into protocol data units and converts them to ATM cells. This is necessary because IP does not recognize conventional MAC-layer protocols, such as those generated on an Ethernet LAN.
- **Local Area Network** (LAN) A system consisting of computer and communications hardware and software connected by a common transmission medium, usually limited to a scope of a few miles.
- **LAN** Local Area Network.
- **LDR** Low Data Rate.
- **LNB** Low-Noise Block.
- **Link** Any physical connection on a network between two separate devices, such as an ATM switch and its associated end point or end station.
- **Megabits Per Second** (Mbps) A digital transmission speed of millions of bits per second.
- **MSE** Mobile Subscriber Equipment.
- **Network-to-Network Interface** (NNI) In an ATM network, the interface between one ATM switch and another, or an ATM switch and a public ATM switching system.
- **NRaD** Naval Command, Control and Ocean Surveillance Center, Research, Development, Training, and Evaluation Division
- **OPNET** Optimized Network Engineering Tools.
- **OSI** Open Systems Interconnection.

- **Packet Switching** A switching technique in which no dedicated path exists between the transmitting device and the receiving device. Information is formatted into individual packets, each with its own address. The packets are sent across the network and reassembled at the receiving station.
- **PCM** Pulse Code Modulation.
- **PIR** Priority Information Requirements.
- **PSH Flag.** This bit, when set, activates the Push Function. The Push function pushes this packet up to the application layer even if the buffer is not yet full.
- **Payload Type Identifier** (PTI) The 3-bit descriptor in an ATM cell header that indicates whether the cell is a user cell or a management cell.
- **Protocol Data Unit** (PDU) A unit of information (e.g., packet or frame) exchanged between peer layers in a network.
- **Permanent Virtual Circuit** (PVC) A generic term for any permanent, provisioned communications medium. NOTE: PVC does not stand for permanent virtual channel. No such term has been defined by any standards organization. Neither has the term "permanent virtual path (PVP)." In ATM, there are two kinds of PVCs: permanent virtual path connections (PVPCs) and permanent virtual channel connections (PVCCs).
- **Physical Layer** The first layer in the OSI Model. It specifies the physical interface (e.g., connectors, voltage levels, cable types) between a user device and the network.
- **Point-to-point** A term used by network designers to describe network links that have only one possible destination for a transmission.
- **Quality of Service** (QoS) The ATM Forum has outlined five categories of performance (Classes 1 through 5) and recommends that ATM's quality of service should be comparable to that of standard digital connections.
- **RF** Radio Frequency.
- **RST Flag.** This bit is set when either side detects problems with the connection and the connection must be reset.
- **Segmentation and Reassembly** (SAR) The process of converting protocol data units into ATM cells (i.e., adjusting the length and format). At the far end, the SAR process takes the payload out of the ATM cells and converts it back into protocol data units.

- **SDR** Surrogate Data Radio.
- **Signaling** (ATM) The procedures used to establish connections on a ATM network. Signaling standards are based on the ITU's Q.93B recommendation.
- **SINCGARS SIP** Single Channel Ground and Airborne Radio System System Improvement Program.
- **SIV** System Integration Van.
- **SPR** Secure Packet Radio.
- **Switch** Device used to route cells through an ATM network.
- **SSCS** Service Specific Convergence Sublayer.
- **Switched Virtual Circuit** (SVC) A generic term for any switched communications medium. NOTE: SVC does not stand for switched virtual channel. No such term has been defined by any standards organization. Neither has the term "switched virtual path (SVP)." In ATM, there are two kinds of SVCs: switched virtual path connections (SVPCs) and switched virtual channel connections (SVCCs).
- **Sustainable Cell Rate** (SCR) A measure of the maximum throughput that can be achieved by bursty traffic over a given virtual connection without the risk of cell loss.
- **Synchronous** A term used to describe a transmission technique that requires a common clock signal (or timing reference) between two communicating devices to coordinate their transmissions. (Compare with "asynchronous.")
- **Synchronous Optical NETwork** (SONET) A set of standards for the digital transmission of information over fiber optics. Based on increments of 51 Mbps.
- **Synchronous Transfer Mode** (STM) In ATM, a method of communications that transmits data streams synchronized to a common clock signal (reference clock). In SDH, it is "Synchronous Transport Module" and is the basic unit (STM-1=155 Mbps, STM-4=622 Mbps, STM-16=2.5Gbps) of the Synchronous Digital Hierarchy.
- **SYN Flag.** A bit marking this packet as a request to establish a connection.
- **TCP** Transmission Control Protocol.
- **TCP HL.** TCP HL stands for TCP Header Length and specifies the total number of bytes contained in the header.

- **TEED** Tactical End-to-End Encryption Device.
- **TI** Tactical Internet.
- **TOC** Tactical Operations Center.
- **TPN** Tactical Packet Network.
- **Traffic Shaping** A mechanism used to control traffic flow so that a specified Quality of Service is maintained.
- **Usage Parameter Control** (UPC) The function of ATM network equipment that controls the Cell Loss Priority bit to control congestion on the network.
- **UIM** Uplink Information Manager.
- **UNI** User-Network Interface.
- **URG Flag**. A bit that, when set, tells the protocol to use the value in the Urgent Pointer.
- **UDP** User Datagram Protocol.
- **User-to-Network Interface** (UNI) ATM Forum standard that defines how private CPE interacts with private ATM switches. A connection that directly links a user's device to an ATM network (usually, through an ATM switch). Also, the physical and electrical demarcation point between the user device and the ATM switch.
- **Variable Bit Rate** (VBR) A type of traffic that, when sent over a network, is tolerant of delays and changes in the amount of bandwidth it is allocated. (e.g., data applications)
- **Virtual Channel Connection** (VCC) A logical communications medium identified by a VCI and carried within a VPC. VCCs may be permanent virtual channel connections (PVCCs), switched virtual channel connections (SVCCs), or smart permanent virtual channel connections (SPVCC). Further, VCC is an end-to-end logical communications medium. Another acronym, VCL (virtual channel link), is more precise, referring to the single segment object identified by a VCI and carried within a VPC. Similarly, a VPC is an end-to-end object and a Virtual Path Link (VPL) is identified a VPI within a link.
- **Virtual Channel Identifier** (VCI) The field in the ATM cell header that labels (identifies) a particular virtual channel.
- **Virtual Circuit** (VC) A generic term for any logical communications medium. NOTE: VC does not stand for virtual channel. Virtual channels are referred

to as VCCs (virtual channel connections). There are three classes of VCs: permanent, switched, and smart (or soft) permanent.

- **Virtual Path Connection** (VPC) A logical communications medium in ATM identified by a virtual path identifier (VPI) and carried within a link. VPCs may be permanent virtual path connections (PVPCs), switched virtual path connections (SVPCs), or smart permanent virtual path connections (SPVPCs). VPCs are uni-directional.
- **VTC** Video Teleconferencing.
- **Virtual Path Identifier** (VPI) The field in the ATM cell header that labels (identifies) a particular virtual path.
- **WIU** Wireless Integrated services digital network interface Unit.
- **WFA** Warfighter Associate.

APPENDIX B. PROTO C CODE FOR STATIC CHANNELS DEFINITION AND ALLOCATION.

```
/*************************************************************************/
/*  Read the channel definition file and define static channels    */
/*  for the ATM switch.                                                  */
/*************************************************************************/
void load_static_channels_definition(temp_atm_state_ptr)
    AtmT_ATM_State* temp_atm_state_ptr;
{
    AtmT_VP_Desc*       temp_vp_ptr;
    AtmT_VC_Desc*       temp_vc_ptr;
    AtmT_Port_Desc*     temp_pdesc_ptr;
    int                   MAX_CHANNELS = 1024;
    int                   MAXSIZE = 11;
    int                   total_ports;
    int                   port_count = 0;
    int                   total_paths;
    int                   path_count = 0;
    int                   total_channels = 0;
    int                   vci_index = 0;
    int                   channel_count = 0;
    int                   value;
    int                   channel_array[1024];
    char                  string[11];
    FILE*                 input_file;

    FIN(load_static_channels_definition(temp_atm_state_ptr));

    /*  Open the channel definition file. */
    if ((input_file = fopen("/usr/work/benton/input_channels", "r"))
                                                          == NULL)
    {
        ams_atm_mgmt_error("Problem opening static channel
                            definition file.", OPC_NIL, OPC_NIL);
    }

    /* Read the first line to get the number of requested channels. */
    if  (fgets ( string, MAXSIZE, input_file ) != NULL)
    {
        if (LTRACE_STATIC_ACTIVE)
```

```
        printf("In load_static_channels function; string is \%s.",
                string);
    if (get_digit(string, &value) == OPC_COMPCODE_FAILURE)
    {
        ams_atm_mgmt_error ("Invalid number of static channels.",
                            OPC_NIL, OPC_NIL);
    }
    total_channels = value;
    if (LTRACE_STATIC_ACTIVE)
    {
        printf("In load_static_channels function after reading ");
        printf("channel count;total_channels = \%d.",
                total_channels);
    }
}

if (total_channels > MAX_CHANNELS)
{
    ams_atm_mgmt_error("Requested virtual channels exceeds allowed
                        maximum.", OPC_NIL, OPC_NIL);
}

/*  Read each channel definition from the file and assign the */
/*  value to the channel array.                               */
while ((fgets ( string, MAXSIZE, input_file ) != NULL) &&
       (channel_count < total_channels))
{
    if (LTRACE_STATIC_ACTIVE)
    {
        printf("In load_static_channels function, reading each ");
        printf(" channel; channel_count = %d.", channel_count);
        printf("In load_static_channels function; string is %s.",
                string);
    }
    if (get_digit(string, &value) == OPC_COMPCODE_FAILURE)
    {
        ams_atm_mgmt_error ("Invalid number for static channels.",
                            OPC_NIL, OPC_NIL);
    }

    if (value > 155000000)
    {
```

```
            ams_atm_mgmt_error ("Exceeded channel capacity in
                        static definition file.", OPC_NIL, OPC_NIL);
        }

        channel_array[channel_count] = value;
        channel_count = channel_count + 1;

        if (LTRACE_STATIC_ACTIVE)
        {
            printf("In load_static_channels function, value ");
            printf("= %d.", value);
        }

    }

    fclose(input_file);

    if (LTRACE_STATIC_ACTIVE)
    {
        printf("In load_static_channels function after closing ");
        printf("file; total_channels = %d.\n", total_channels);
    }

    total_ports = temp_atm_state_ptr->port_data.port_count;

    if (LTRACE_STATIC_ACTIVE)
    {
        printf("In load_static_channels function; ");
        printf("total_ports = \%d.", total_ports);
    }

    port_count = 0;
    while (port_count < total_ports)
    {
        temp_pdesc_ptr =
            &temp_atm_state_ptr->port_data.port_array[port_count];
        total_paths = temp_pdesc_ptr->vp_data.vp_count;

        if (LTRACE_STATIC_ACTIVE)
        {
            printf("In load_static_channels function; ");
            printf("total_paths = %d.\n", total_paths);
```

```
        }

        path_count = 0;

        while (path_count < total_paths)
        {
            temp_vp_ptr =
                   &temp_pdesc_ptr->vp_data.vp_array[path_count];
            if (temp_vp_ptr->qos_desc.class == 3)
            {
                ams_atm_vc_elements_add(temp_vp_ptr,
                                        total_channels);

                if (LTRACE_STATIC_ACTIVE)
                {
                    printf("In load_static_channels function; ");
                    printf("after adding channels total_channels =
                           %d.\n", total_channels);
                }

                channel_count = 0;

                while (channel_count < total_channels)
                {
                  /*  Assign the requested bandwidth to the channel */
                  temp_vp_ptr->vc_array
                      [channel_count].alloc_bandwidth_in
                                    = channel_array[channel_count];
                  temp_vp_ptr->vc_array
                      [channel_count].alloc_bandwidth_out
                                    = channel_array[channel_count];
                  channel_count = channel_count + 1;
                }
            }

            path_count = path_count + 1;

        }

        port_count = port_count + 1;
    }
```

```
    /*  User did not provide the correct number of channel          */
    /*  definitions.                                                */
    if (channel_count != total_channels)
    {
        ams_atm_mgmt_error ("Did not define correct number of
                             channels", OPC_NIL, OPC_NIL);
    }

    FOUT;
}


/*****************************************************************/
/*  Verify the character string contains only numeric digits and  */
/*  then convert the character string to a number.                */
/*****************************************************************/

Compcode get_digit (input_string, value)
    char* input_string;
    int*  value;
{
    #define  YES  1
    #define  NO   0

    char  ch;
    char  number[11];
    int   digit = YES;
    int   count = 0;

    FIN(get_digit(input_string, value));

/*  Remove the newline character from the input string           */
    while ((ch=input_string[count]) != '\n')
    {
        number[count] = input_string[count];
        count = count + 1;
    }

    number[count] = '\0';

    count = 0;
```

```
/*  Check each character in the input string for a numeric digit.  */
/*  If any character fails the test, set the flag to NO.           */
    while ((ch=number[count]) != '\0' && digit == YES)
    {
        if (!isdigit(ch))
        {
            digit = NO;
}

        count = count + 1;

    }

/*  If all the characters were numeric digits, then convert the    */
/*  string to a number and return SUCCESS.  Otherwise, return      */
/*  FAILURE.                                                       */
    if (digit == YES)
    {
        *value = atoi(input_string);
        if (LTRACE_STATIC_ACTIVE)
            printf("In get_digit; value = \%d.", *value);

        FRET(OPC_COMPCODE_SUCCESS);
    }
    else
        FRET(OPC_COMPCODE_FAILURE);
}


/*************************************************************************/
/*  This function finds a valid vitrual path and channel given a   */
/*  specific atm switch and port via atm_state_ptr and port_value. */
/*  respectfully.                                                  */
/*************************************************************************/

Compcode ams_atm_avail_static_vp_vc_find (atm_state_ptr,
                                          port_value, vpi_value_ptr,
                                          vci_value_ptr,
                         traf_con_ptr, qos_class)
    AtmT_ATM_State*      atm_state_ptr;
    int                   port_value;
    int*                  vpi_value_ptr;
```

```
    int*                    vci_value_ptr;
    AmsT_Traf_Contract*     traf_con_ptr;
    int                     qos_class;
{
    AtmT_Port_Desc*         pdesc_ptr;
    AtmT_VP_Desc*           vp_ptr;
    AtmT_VC_Desc*           vc_ptr;
    int                     vpi_index;
    int                     vci_index;

    /* This procedure finds a VPI and VCI value for a channel in   */
    /* the specified direction in the specified port.  If there    */
    /* is no VP with sufficient available bandwidth and correct    */
    /* Qos class, then a failure status is returned; otherwise,    */
    /* success.  No state information is modified concerning the   */
    /* VP and VCs.                                                 */
    FIN (ams_atm_avail_static_vp_vc_find (<args>));

    /* Obtain the port desc pointer.                               */
    pdesc_ptr = &atm_state_ptr->port_data.port_array [port_value];

    /* Loop through the VPs on the port and find one with          */
    /* sufficient bandwidth.           */
    for (vpi_index = 0; vpi_index < pdesc_ptr->vp_data.vp_count;
         vpi_index++)

        /* Do not use the signalling VP.                           */
        if (vpi_index == AMSC_ATM_SIGVP_VPI)
            continue;

        /* Obtain the vpi_index-th in_VP pointer.            */
        vp_ptr = &pdesc_ptr->vp_data.vp_array [vpi_index];

        /* Determine if this VP has the correct QoS class.         */
        if (vp_ptr->qos_desc.class != qos_class)
        {
            /* This VP does not support the requested QoS class;   */
            /* continue to the next VP.                            */
            vp_ptr = OPC_NIL;
            continue;
        } /* End if vp_ptr->qos_desc.class != qos_class            */
```

```
/* Determine if this VP can support the traffic.          */
        if (ams_atm_vp_supports_traffic (atm_state_ptr,
                                port_value, vpi_index,
                                traf_con_ptr) == OPC_TRUE)
        {
            /* This VP has sufficient bandwidth.                */
            /* Break out of loop.                               */
            *vpi_value_ptr = vpi_index;
            break;
        } /* End if ams_atm_vp_supports_traffic                 */

        /* This VP has insufficient bandwidth. Try the next one.  */
        /* Set the pointer to NIL, so that it can verified if the */
        /* VP search failed.                                      */
        vp_ptr = OPC_NIL;
    } /* End for loop                                           */

    /* If the VP pointer is NIL, then the search failed.        */
    /* Handle failure.                                          */
    if (vp_ptr == OPC_NIL)
    {
        FRET (OPC_COMPCODE_FAILURE);
    } /* End if vp_ptr == OPC_NIL                               */

    /* Found an available VP now search for an available VC.    */
    for (vci_index = 0; vci_index < vp_ptr->vc_count; vci_index++)
    {
        /* Obtain the vci_index-th in VC pointer.               */
        vc_ptr = &vp_ptr->vc_array [vci_index];

        /* Determine if the VC is free.                         */
        if (vc_ptr->status == AMSC_VC_FREE)
        {
            /* Determine if this VC can support the traffic.    */
            if (ams_atm_vc_supports_traffic (atm_state_ptr,
                    port_value, vpi_index, traf_con_ptr, vci_index)
                                                    == OPC_TRUE)
            {
                /* This VP has sufficient bandwidth.            */
                /* Break out of loop.                           */
                *vpi_value_ptr = vpi_index;
                break;
```

```
            } /* End if ams_atm_vc_supports_traffic                */
        } /* End if vc_ptr->status == AMSC_VC_FREE

        /* The VC is in use.  Set the pointer to NIL.               */
        vc_ptr = OPC_NIL;
    }

    /* If the VC pointer is NIL, then the search failed.            */
    /* Handle failure.                                              */
    if (vc_ptr == OPC_NIL)
    {
        FRET (OPC_COMPCODE_FAILURE);
    }
    else
    {
        FRET (OPC_COMPCODE_SUCCESS);
    }
} /* End function ams_atm_avail_static_vp_vc_find                  */


/*****************************************************************/
/*  This function allocates a specific port, virtual path and    */
/*  virtual channel to a call.                                   */
/*****************************************************************/

AtmT_VC_Desc* ams_atm_static_vp_vc_alloc (atm_state_ptr,
                  port_value, vpi_value, vci_value, traf_con_ptr)
    AtmT_ATM_State*     atm_state_ptr;
    int                   port_value;
    int                   vpi_value;
    int                   vci_value;
    AmsT_Traf_Contract* traf_con_ptr;
{
    AtmT_Port_Desc*     pdesc_ptr;
    AtmT_VP_Desc*       vp_ptr;
    AtmT_VC_Desc*       vc_ptr;
    int                   vc_add_count;

    /** This procedure allocates the VPI and VCI to the call and   */
    /** adjusts the available bandwidth values.  The pointer to    */
    /** the specified VC ds is returned.                           */
    FIN (ams_atm_static_vp_vc_alloc (<args>));
```

```
    /* Verify that the index values are valid.                          */
    ams_atm_static_port_vp_vc_verify (atm_state_ptr,
                                      port_value, vpi_value, vci_value);

    /* Obtain pointers to the port, VP and VC data structures.          */
    pdesc_ptr = &atm_state_ptr->port_data.port_array [port_value];
    vp_ptr = &pdesc_ptr->vp_data.vp_array [vpi_value];
    vc_ptr = &vp_ptr->vc_array [vci_value];

    /* Verify that the VC is not in use.                                */
    if (vc_ptr->status == AMSC_VC_IN_USE)
        ams_atm_error ("VC is already in use.  Unable to allocate VC
                        for call.", OPC_NIL, OPC_NIL);

    /* Set VP's fields.                                                 */
    vp_ptr->avail_bandwidth_in -=
            traf_con_ptr->called_ctd.src_traf_desc.pcr *
                                            AMSC_ATM_CELL_SIZE;
    vp_ptr->avail_bandwidth_out -=
            traf_con_ptr->calling_ctd.src_traf_desc.pcr *
                                            AMSC_ATM_CELL_SIZE;
    vp_ptr->avail_vc_count--;

    /* Set VC's fields.                                                 */
    vc_ptr->status = AMSC_VC_IN_USE;

    FRET (vc_ptr);
}/* end function ams_atm_static_vp_vc_alloc                            */


/*********************************************************************/
/*  This function will verify the port_value, vpi_value and          */
/*  vci_value are all valid for the atm switch defined by the        */
/*  atm_state_ptr.                                                   */
/*********************************************************************/

void ams_atm_static_port_vp_vc_verify (atm_state_ptr,
                                       port_value, vpi_value, vci_value)
    AtmT_ATM_State*  atm_state_ptr;
    int              port_value;
    int              vpi_value;
```

```
    int                 vci_value;
{
    AtmT_Port_Desc*  pdesc_ptr;
    AtmT_VP_Desc*    vp_ptr;
    char*              attempt_msg = "Attempted to determine use of
                                      VP/VC on port.";

    /** This procedure verifies that the port, VPI, and VCI values  */
    /** are valid.  If not, then an error message is displayed and  */
    /** the simulation is terminated.                               */
    FIN (ams_atm_static_port_vp_vc_verify (args));

    if (port_value < 0)
    {
        ams_atm_error (attempt_msg, "Port value provided is negative,
                       which is not a valid value.", OPC_NIL);
    }
    else if (vpi_value < 0)
        {
            ams_atm_error (attempt_msg, "VPI value provided is
                  negative, which is not a valid value.", OPC_NIL);
        }
        else if (vci_value < 0)
            {
                ams_atm_error (attempt_msg, "VCI value provided
                          is negative, which is not a valid value.",
                            OPC_NIL);
            }

    if (port_value >= atm_state_ptr->port_data.port_count)
    {
        ams_atm_error (attempt_msg, "Port value provided is greater
                       than any actual port.", OPC_NIL);
    }
    else
        pdesc_ptr = &atm_state_ptr->port_data.port_array
                                                    [port_value];

    if (vpi_value >= pdesc_ptr->vp_data.vp_count)
    {
        ams_atm_error (attempt_msg, "VPI value provided is
                       greater than any actual VPI within the port.",
```

```
                    OPC_NIL);
    }
    else
        vp_ptr = &pdesc_ptr->vp_data.vp_array [vpi_value];

    if (vci_value >= vp_ptr->vc_count)
    {
        /* The requested VC is greater than any in the VP.          */
        ams_atm_error (attempt_msg, "VCI value provided is greater
                    than any actual VCI within the VPI.",
                    OPC_NIL);
    }

    FOUT;
}/* end function ams_atm_static_port_vp_vc_verify                   */


/*****************************************************************/
/*  This function verifies the virtual channel can support the   */
/*  required traffic load.                                       */
/*****************************************************************/

Boolean ams_atm_vc_supports_traffic (atm_state_ptr, port_value,
                                vpi_value, traf_con_ptr, vci_value)
    AtmT_ATM_State*      atm_state_ptr;
    int                    port_value;
    int                    vpi_value;
    int                    vci_value;
    AmsT_Traf_Contract*  traf_con_ptr;
{
    AtmT_Port_Desc*      pdesc_ptr;
    AtmT_VP_Desc*        vp_ptr;
    AtmT_VC_Desc*        vc_ptr;
    Boolean                supports_traffic;
    double                 in_pcr;
    double                 out_pcr;

    /* Determines if the VC can accomodate the traffic             */
    /* requirements.  If so, OPC_TRUE is returned; else, OPC_FALSE */
    FIN (ams_atm_vc_supports_traffic (<args>));

    /* Cache the incoming and outgoing Peak Cell Rates (PCRs) as    */
```

```
    /* defined by the traffic contract.                                    */
    in_pcr  = traf_con_ptr->called_ctd.src_traf_desc.pcr;
    out_pcr = traf_con_ptr->calling_ctd.src_traf_desc.pcr;

    /* Obtain the port, VP and VC data structure pointers.                 */
    pdesc_ptr = &atm_state_ptr->port_data.port_array [port_value];
    vp_ptr = &pdesc_ptr->vp_data.vp_array [vpi_value];
    vc_ptr = &vp_ptr->vc_array[vci_value];

    /* If the rate is less than that available in the VC, then the         */
    /* VC is available.                                                    */
    if ((vc_ptr->alloc_bandwidth_in > (in_pcr *
            AMSC_ATM_CELL_SIZE)) && (vc_ptr->alloc_bandwidth_out
            > (out_pcr * AMSC_ATM_CELL_SIZE)))
    {
        /* The VC meets the requirements.                                  */
        supports_traffic = OPC_TRUE;
    }
    else
    {
        /* The VP/VC does not meet the requirements.                       */
        supports_traffic = OPC_FALSE;
    }

    FRET (supports_traffic);

} /* end function ams_atm_vc_supports_traffic                             */
```

APPENDIX C. BADD_CALL_REQUESTOR.

...
...

External File Set

attribute	*value*	*type*	*default value*
external file set	badd_functions	typed file	

Process Model Comments

General Process Description:

The process initiates call requests and sends them to the badd_call_scheduler module.

ICI Interfaces:

---

badd_call_req_if_ici

Packet Formats:

None.

Statistic Interfaces:

None

Process Registry:

Published Attributes: None
Expected Attributes: None

Restrictions:

None

Process Model Attributes

Attribute **interarrival time** properties

Property	*Value*
Default Value:	0.001
Data Type:	double
Attribute Description:	Private
Auto. assign value:	FALSE
Units:	seconds
Low Range:	0.0 exclusive
Comments:	Specifies the interarrival time between packets.

Attribute **packet size** properties

Property	*Value*
Default Value:	1000
Data Type:	integer
Attribute Description:	Private
Auto. assign value:	FALSE

Units:	bits
Comments:	Specifies the size of the packets generated (in bits).

Attribute **call wait time** properties	
Property	*Value*
Default Value:	10
Data Type:	double
Attribute Description:	Private
Auto. assign value:	FALSE
Units:	seconds
Low Range:	0.0 exclusive
Comments:	Specifies the time between calls.

Attribute **call duration** properties	
Property	*Value*
Default Value:	1,000
Data Type:	double
Attribute Description:	Private
Auto. assign value:	FALSE
Units:	seconds
Low Range:	0.0 exclusive
Comments:	Specifies the length of a call.

Attribute **dest addr** properties		
Property	*Value*	
Default Value:	NONE	
Data Type:	integer	
Attribute Description:	Private	
Auto. assign value:	FALSE	
Comments:	Specifies the destination of the call.	
Symbol Map:	*Symbol*	*Value*
	NONE	-1
Allow other values:	YES	

Attribute **AAL type** properties		
Property	*Value*	
Default Value:	5	
Data Type:	integer	
Attribute Description:	Private	
Auto. assign value:	FALSE	
Comments:	Specifies the AAL type to be used.	
Symbol Map:	*Symbol*	*Value*
	1	1
	2	2
	34	34

	5	5
Allow other values:	NO	

Attribute **QoS class** properties		
Property	*Value*	
Default Value:	D	
Data Type:	string	
Attribute Description:	Private	
Auto. assign value:	FALSE	
Comments:	Specifies the QoS class.	
Symbol Map:	*Symbol*	*Value*
	A	A
	B	B
	C	C
	D	D
Allow other values:	NO	

Process Model Interface Attributes		
Attribute **begsim intrpt** properties		
Property	*Value*	*Inherit*
Assign Status:	hidden	
Initial Value:	enabled	N/A
Default Value:	disabled	YES
Data Type:	toggle	N/A
Attribute Description:	Private	N/A
Comments:	This attribute specifies whether a 'begin simulation interrupt' is generated for a processor module's root process at the start of the simulation.	YES
Symbol Map:	NONE	YES

Attribute **endsim intrpt** properties		
Property	*Value*	*Inherit*
Assign Status:	hidden	
Initial Value:	disabled	N/A
Default Value:	disabled	YES
Data Type:	toggle	N/A
Attribute Description:	Private	N/A
Comments:	This attribute specifies whether an 'end simulation interrupt' is generated for a processor module's root process at the end of the simulation.	YES
Symbol Map:	NONE	YES

Attribute **failure intrpts** properties		
Property	*Value*	*Inherit*
Assign Status:	hidden	
Initial Value:	disabled	N/A

Default Value:	disabled	YES
Data Type:	enumerated	N/A
Attribute Description:	Private	N/A
Comments:		YES
	This attribute specifies whether failure interrupts are generated for a processor module's root process upon failure of nodes or links in the network model.	
Symbol Map:	NONE	YES

Attribute **intrpt interval** properties

Property	*Value*	*Inherit*
Assign Status:	hidden	
Initial Value:	disabled	N/A
Default Value:	disabled	YES
Data Type:	toggle double	N/A
Attribute Description:	Private	N/A
Units:	sec.	YES
Comments:		YES
	This attribute specifies how often regular interrupts are scheduled for the root process of a processor module.	
Symbol Map:	NONE	YES

Attribute **priority** properties

Property	*Value*	*Inherit*
Assign Status:	hidden	
Initial Value:	0	N/A
Default Value:	0	YES
Data Type:	integer	N/A
Attribute Description:	Private	N/A
Low Range:	-32767 inclusive	YES
High Range:	32767 inclusive	YES
Comments:		YES
	This attribute is used to determine the execution order of events that are scheduled to occur at the same simulation time.	
Symbol Map:	NONE	YES

Attribute **recovery intrpts** properties

Property	*Value*	*Inherit*
Assign Status:	hidden	
Initial Value:	disabled	N/A
Default Value:	disabled	YES
Data Type:	enumerated	N/A
Attribute Description:	Private	N/A
Comments:		YES
	This attribute specifies whether recovery interrupts are scheduled for the processor module's root process upon recovery of nodes or links in the network model.	
Symbol Map:	NONE	YES

...
...

Attribute **super priority** properties

Property	*Value*	*Inherit*
Assign Status:	hidden	
Initial Value:	disabled	N/A
Default Value:	disabled	YES
Data Type:	toggle	N/A
Attribute Description:	Private	N/A
Comments:	This attribute is used to determine the execution order of events that are scheduled to occur at the same simulation time.	YES
Symbol Map:	NONE	YES

Process Model Simulation Attributes

Attribute **class_A_CDV_tolerance** properties

Property	*Value*
Default Value:	0.0
Data Type:	double
Attribute Description:	Private
Auto. assign value:	FALSE
Comments:	Upper bound on cell delay variance that two consecutive Quality of Service (QoS) class A cells may experience.

Attribute **class_B_CDV_tolerance** properties

Property	*Value*
Default Value:	0.0
Data Type:	double
Attribute Description:	Private
Auto. assign value:	FALSE
Comments:	Upper bound on cell delay variance that two consecutive Quality of Service (QoS) class B cells may experience.

Attribute **class_C_CDV_tolerance** properties

Property	*Value*
Default Value:	0.0
Data Type:	double
Attribute Description:	Private
Auto. assign value:	FALSE
Comments:	Upper bound on cell delay variance that two consecutive Quality of Service (QoS) class C cells

may experience.

Attribute **class_D_CDV_tolerance** properties

Property	*Value*
Default Value:	0.0
Data Type:	double
Attribute Description:	Private
Auto. assign value:	FALSE
Comments:	Upper bound on cell delay variance that two consecutive Quality of Service (QoS) class D cells may experience.

Header Block

```
#include "/usr/local/mil3_dir/3.0.A_DL5/models/std/atm/ams_interfaces.h"
#include "/usr/local/mil3_dir/3.0.A_DL5/models/std/atm/ams_aal_interfaces.h"
/* Code added for badd */
#include "/usr/work/benton/op_code/badd_interface.h"
/* end code added for badd */

/* These are transition conditions. */
#define SIGNAL                  ((op_intrpt_type () == OPC_INTRPT_REMOTE) &&     \
                                 (op_intrpt_code () == AMSC_INTERFACE_SIGNAL))

#define EST_CON                 (SIGNAL && (primitive == AMSC_AAL_ESTAB_Con))

#define REL_IND                 (SIGNAL && (primitive == AMSC_AAL_RELEASE_Ind))

#define REL_CON                 (SIGNAL && (primitive == AMSC_AAL_RELEASE_Con))

#define CALL_START              ((op_intrpt_type () == OPC_INTRPT_SELF) &&       \
                                 (op_intrpt_code () == AMSC_TGEN_CALL_START))

#define CALL_END                ((op_intrpt_type () == OPC_INTRPT_SELF) &&       \
                                 (op_intrpt_code () == AMSC_TGEN_CALL_END))

#define NEIGHBOR_NOTIFY         ((op_intrpt_type () == OPC_INTRPT_REMOTE) &&     \
                                 (op_intrpt_code () == AMSC_NEIGHBOR_NOTIFY))

#define NOTIFY_COMPLETE         (ams_neighbor_notify_is_complete (nbr_data_ptr) == OPC_TRUE)

/* Code added for badd */
#define WAIT_CALL_DURATION  ((op_intrpt_type () == OPC_INTRPT_SELF) &&           \
                             (op_intrpt_code () == AMSC_TGEN_WAIT_CALL_DURATION))
/* End code added for badd */

/* These are the ams_traf_gen self interrupt codes. */
#define  AMSC_TGEN_CALL_START   0
#define AMSC_TGEN_CALL_END   1
#define AMSC_TGEN_DATA_GEN   2
/* Code added for badd */
#define AMSC_TGEN_WAIT_CALL_DURATION   3
/* End code added for badd */
```

```
40
   /* The ams_traf_gen process will output trace information if this conditional is true. */
   #define LTRACE_ACTIVE  (debug_mode &&                                    \
                           (op_prg_odb_ltrace_active ("ams") ||             \
45                          op_prg_odb_ltrace_active ("ams_traf") ||         \
                            op_prg_odb_ltrace_active ("ams_traf_gen")))

   #define LTRACE_CONNECT_ACTIVE    (op_prg_odb_ltrace_active ("connect"))

50 /* Code added for badd */
   #define LTRACE_CALL_REQUESTOR_ACTIVE   (op_prg_odb_ltrace_active ("call_requestor"))

   /* Procedure declarations. */
   void badd_call_req_nbr_intrpt_proc ();
55 void badd_call_req_spurious_signal_handle ();
   void badd_call_req_error ();

60

65
```

State Variable Block

```
   int                  \debug_mode;
   int                  \packet_size;
   Distribution*        \int_arrival_distptr;
   Distribution*        \call_wait_distptr;
5  Distribution*        \call_duration_distptr;
   double               \avg_rate;
   Objid                \aal_module_id;
   Ici*                 \aal_handle_iciptr;
   Evhandle             \next_packet_arrival;
10 Evhandle             \call_end_intrpt;
   int                  \dest_addr;
   char                 \pid_string [128];
   int                  \to_aal_stream_index;
   Objid                \my_id;
15 int                  \AAL_type;
   int                  \qos_class;
   AmsT_Traf_Contract*  \traf_con_ptr;
   AmsT_Neighbor_Data*  \nbr_data_ptr;

20 /* Code added for badd */
   double           \int_arr_time;
   double           \call_wait_time;
   double           \call_duration;
   Objid            \sch_module_id;
25 int              \to_sch_stream_index;
   Badd_Sch_Mod_Data    \Scheduler_Module;
   Badd_Sch_Call_Desc   \Scheduler_Call;
```

```
/* End code added for badd */
```

Temporary Variable Block

```
int                 primitive;
Packet*             pkptr;
Ici*                if_iciptr;
AmsT_Traf_Contract* tmp_traf_con_ptr;
double              peak_cell_rate;
char                qos_class_string [128];
double              cdv_tolerance;

/* Code added for badd */
int                 cd_packet_size;
int                 clark_dest_addr;
double      badd_packet_size;
double              event_time;
double              end_sim_time;
extern int          total_calls_requested;
Prohandle   call_gen_prohandle;
/* End code added for badd */
```

Function Block

```
void
badd_call_req_nbr_intrpt_proc (ndata_ptr, ndesc_ptr, state_ptr)
    AmsT_Neighbor_Data* ndata_ptr;
    AmsT_Neighbor_Desc* ndesc_ptr;
    void*               state_ptr;
    {
    AmsT_Neighbor_Verify_Desc   vdesc;
    int                         to_sw_stream_index;

    /** This procedure handles a neighbor notification event in an AMS      **/
    /** traf gen specific manner. It determines the neighbor's object       **/
    /** ID and type, and verifies that there are a correct number of        **/
    /** interconnections.                                                   **/
    FIN (badd_call_req_nbr_intrpt_proc (ndata_ptr, ndesc_ptr, state_ptr));

    /* Switch based on the AMS type.                                        */
    switch (ndesc_ptr->module_amstype)
        {
        case AMSC_MTYPE_AAL_CLIENT:
            {
            /* Build the verify desc data structure.                        */
            vdesc.mod_id        = my_id;
            vdesc.nbr_id        = ndesc_ptr->module_objid;
            vdesc.nbr_id_ptr    = &sch_module_id;
            vdesc.mod_name      = "BADD Call Requestor";
            vdesc.nbr_name      = "call_sch";
            vdesc.nbr_type      = AMSC_MTYPE_AAL_CLIENT;
```

```
                vdesc.from_nbr_strm_cnt         = 1;
                vdesc.from_nbr_strm_index_ptr   = OPC_NIL;
                vdesc.to_nbr_strm_cnt           = 1;
                vdesc.to_nbr_strm_index_ptr     = &to_sch_stream_index;

                /* Verify that the neighbor has the correct                */
                /* characteristics.                                        */
                ams_neighbor_verify (&vdesc);

                break;
                }

            default:
                {
                /* This is an unexpected neighbor notification.            */
                /* Issue error message.                                    */
                op_sim_end ("Process received a neighbor notification from a module",
                    "of an unexpected type.  Simulation terminated.", OPC_NIL, OPC_NIL);

                break;
                }
            }

        FOUT;
        }

void
badd_call_req_spurious_signal_handle ()
    {
    Ici*        if_iciptr;
    Ici*        release_if_iciptr;
    int         primitive;
    Ici*        ll_handle_iciptr;
    Packet*     pkptr;

    /* This procedure handles spurious interrupts.                         */
    /* Only three types of spurious interrupts can be accepted. All        */
    /* others must result in an op_sim_end (). These three                 */
    /* interrupts are:                                                     */
    /*   1. An AAL ESTAB Ind.                                              */
    /*   2. An AAL RELEASE Con.                                            */
    /*   3. A packet arrival.                                              */
    FIN (badd_call_req_spurious_signal_handle ());

    if (SIGNAL)
        {
        /* This is a signal from the AAL.                                  */

        /* Obtain the ICI pointer.                                         */
        if_iciptr = op_intrpt_ici ();

        /* Obtain the primitive from the ICI.                              */
        op_ici_attr_get (if_iciptr, "primitive", &primitive);

        /* Obtain the 'lower layer handle' from the ICI.                   */
        op_ici_attr_get (if_iciptr, "lower layer handle", &ll_handle_iciptr);

        /* Switch on the 'primitive'.          */
        switch (primitive)
```

```
                {
                case AMSC_AAL_ESTAB_Ind:
                    {
                    /* This is an 'establish indication' from the AAL.          */

                    if (LTRACE_ACTIVE)
                        {
                        op_prg_odb_print_major (pid_string, "Received spurious AAL ESTABLISH Request signal.",
                            "Call not accepted.", "Sending AAL RELEASE Request signal.", OPC_NIL);
                        }

                    /* Set the 'upper layer handle' in the interface            */
                    /* ICI for the 'establish indication' signal, since         */
                    /* the lower layer expects it to be filled in. The          */
                    /* ICI is not destroyed, since this is a forced             */
                    /* interrupt and the lower layer process expects            */
                    /* to obtain the handle from the ICI when the               */
                    /* control returns.                                         */
                    op_ici_attr_set (if_iciptr, "upper layer handle", OPC_NIL);

                    /* Simply send an AAL_RELEASE_Req to request that           */
                    /* the connection be released.                              */
                    release_if_iciptr = op_ici_create (AMSC_INTERFACE_ICI);
                    op_ici_attr_set (release_if_iciptr, "primitive", AMSC_AAL_RELEASE_Req);
                    op_ici_attr_set (release_if_iciptr, "lower layer handle", ll_handle_iciptr);
                  op_ici_install (release_if_iciptr);

                    /* Send a remote interrupt which will carry the ICI to the AAL module. */
                    op_intrpt_schedule_remote (op_sim_time (), AMSC_INTERFACE_SIGNAL, aal_module_id);

                    break;
                    }

                case AMSC_AAL_RELEASE_Con:
                    {
                    /* This is a 'release confirm' from the AAL.                */

                    if (LTRACE_ACTIVE)
                        {
                        op_prg_odb_print_major (pid_string, "Received spurious AAL RELEASE Confirm signal.",
                            "This is in response to AAL RELEASE Request terminating spurious connection.
                        }

                    /* This is a response to our earlier 'release               */
                    /* request' which was a response to the                     */
                    /* spurious 'estab indication'.                             */
                    /* Do nothing other than destroy the ICI.                   */
                    op_ici_destroy (if_iciptr);

                    break;
                    }

                default:
                    {
                    /* This is some other completely unexpected                 */
                    /* signal. Issue error message and terminate                */
                    /* simulation.                                              */
                    op_sim_end ("Received unexpected signal.", "", "", "");
                    }
```

```
            }
        }
    else if (op_intrpt_type () == OPC_INTRPT_STRM)
        {
150     /* This is a spurious packet arrival. The ams_traf_gen          */
        /* just destroys this packet.                                   */
        if (LTRACE_ACTIVE)
            {
            op_prg_odb_print_major (pid_string, "Received spurious DATA packet.",
155             "Destroying the packet.", OPC_NIL);
            }

        /* Get the packet fromt the stream and destroy it.              */
        pkptr = op_pk_get (op_intrpt_strm ());
160     op_pk_destroy (pkptr);
        }
    else
        {
        /* This is some other completely unexpected                     */
165     /* interrupt. Issue error message and terminate                 */
        /* simulation.                                                  */
        op_sim_end ("Received unexpected interrupt.", "", "", "");
        }

170 FOUT;
    }

void
badd_call_req_error (msg0, msg1, msg2)
175     char*       msg0;
        char*       msg1;
        char*       msg2;
        {
        /** Print an error message and exit the simulation. **/
180     FIN (badd_call_reg_error (msg0, msg1, msg2));

        op_sim_end ("Error in badd_call_requestor process:",
            msg0, msg1, msg2);

185     FOUT;
        }
```

Diagnostic Block

```
/* Display connection information, if requested. */
if (LTRACE_CONNECT_ACTIVE)
    {
5   /* Print information. */
    ams_neighbor_data_print (nbr_data_ptr, ams_neighbor_desc_print_noop);
    }
```

...
...

forced state **INIT**

attribute	*value*	*type*	*default value*
name	INIT	string	st
enter execs	(See below.)	textlist	
exit execs	(empty)	textlist	(empty)
status	forced	toggle	unforced

enter execs **INIT**

```
   /* Determine the initial values of the state variables and set up          */
   /* the initial state of this instantiation of the ams_traf_gen             */
   /* process.                                                                */

 5 /* Obtain the object ID of this process' parent module.                    */
   my_id = op_id_self ();

   /* Obtain the attribute values for the packet interarrival                 */
   /* time, size, wait time between calls, call duration, and                 */
10 /* destination address. These values are process attributes.               */
   op_ima_obj_attr_get (my_id, "interarrival time", &int_arr_time);
   op_ima_obj_attr_get (my_id, "packet size", &packet_size);
   op_ima_obj_attr_get (my_id, "call wait time", &call_wait_time);
   op_ima_obj_attr_get (my_id, "call duration", &call_duration);
15 op_ima_obj_attr_get (my_id, "dest addr", &dest_addr);
   op_ima_obj_attr_get (my_id, "QoS class", qos_class_string);
   op_ima_obj_attr_get (my_id, "AAL type", &AAL_type );

   /* Convert the Quality of Service class character into                     */
20 /* the index value and check its validity.                                 */
   ams_qos_class_char_to_index_convert (qos_class_string [0], &qos_class);
   if (qos_class == AMSC_QOS_CLASS_UNDEF)
       {
       /* The QoS Class value is invalid. Issue error message                */
25     /* and terminate the simulation.                                      */
       op_sim_end ("Specified Quality of Service Class attribute value is invalid.",
           "Value must be 'A', 'B', 'C', or 'D'.", "", "");
       }

30 /* Load in the call wait distribution */
   call_wait_distptr = op_dist_load("constant", call_wait_time, 0.0);
   call_duration_distptr = op_dist_load ("constant", call_duration, 0.0);

   /* determine whether or not the simulation is in debug mode.               */
35 debug_mode = op_sim_debug ();

   /* Initialize the AAL module object ID to NULL value.                      */
   aal_module_id = OPC_OBJID_NULL;
   sch_module_id = OPC_OBJID_NULL;
40
   /* Generate PID display string.                                            */
   sprintf (pid_string, "badd_call_requestor PID (%d)", op_pro_id (op_pro_self ()));

   /* Obtain and send out neighbor information.                               */
45 nbr_data_ptr = ams_neighbor_data_build ();

   ams_neighbor_notify (nbr_data_ptr, BADD_MTYPE_SCHEDULER_CLIENT);
```

```
/* Code added by BADD */
if (LTRACE_CALL_REQUESTOR_ACTIVE)
{
  printf("In badd_call_requestor.init: printing badd_call_requestor neighbor info.\n");
/*  ams_neighbor_data_print (nbr_data_ptr, ams_neighbor_desc_print_noop);
/*  badd_call_req_error ("In badd_call_requestor.init: ending simulation test.\n"); */
} /* End if (LTRACE_CALL_REQUESTOR_ACTIVE) */

if (LTRACE_CALL_REQUESTOR_ACTIVE)
{
  printf("In badd_call_requestor.init state: Leaving init state. \n");
}

/* End code added for BADD */
```

transition **INIT -> config**

attribute	*value*	*type*	*default value*
name	tr_1	string	tr
condition		string	
executive		string	
color	RGB333	color	RGB333
drawing style	line	toggle	spline

unforced state **config**

attribute	*value*	*type*	*default value*
name	config	string	st
enter execs	(empty)	textlist	(empty)
exit execs	(See below.)	textlist	
status	unforced	toggle	unforced

exit execs **config**

```
/* Ams_traf_gen expects either a neighbor notification interrupt,          */
/* or a spurious signal.                                                   */
if (NEIGHBOR_NOTIFY)
    {
    /* This is a 'neighbor notify' signal.                                 */
    if (LTRACE_ACTIVE)
        {
        op_prg_odb_print_major (pid_string, "Received neighbor notification.", OPC_NIL);
        }

    /* Handle the neighbor notification.                                   */
    ams_neighbor_interrupt_handle (nbr_data_ptr, badd_call_req_nbr_intrpt_proc, OPC_NIL);
    }
else
    {
    /* This is a spurious interrupt.  Handle appropriately.                */
```

```
    badd_call_req_spurious_signal_handle ();
    }
```

transition **config -> config**

attribute	*value*	*type*	*default value*
name	tr_0	string	tr
condition	default	string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

transition **config -> schedule**

attribute	*value*	*type*	*default value*
name	tr_2	string	tr
condition	NOTIFY_COMPLETE	string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

forced state **schedule**

attribute	*value*	*type*	*default value*
name	schedule	string	st
enter execs	(See below.)	textlist	
exit execs	(empty)	textlist	(empty)
status	forced	toggle	unforced

enter execs **schedule**

```
    /* Code added for BADD          */

    event_time = op_sim_time() + op_dist_outcome (call_wait_distptr);

 5  if (LTRACE_CALL_REQUESTOR_ACTIVE)
    {
      printf("In clark_badd_call_requestor.schedule; schedule start for %f.\n",
          event_time);
    }
10  /* End code added for BADD  */

    /* Start call by scheduling self intrpt. */
    op_intrpt_schedule_self (event_time, AMSC_TGEN_CALL_START);

15
```

...
...

transition **schedule -> wait_call_idle**

attribute	*value*	*type*	*default value*
name	tr_3	string	tr
condition		string	
executive		string	
color	RGB333	color	RGB333
drawing style	line	toggle	spline

unforced state **wait_call_idle**

attribute	*value*	*type*	*default value*
name	wait_call_idle	string	st
enter execs	(empty)	textlist	(empty)
exit execs	(See below.)	textlist	
status	unforced	toggle	unforced

exit execs **wait_call_idle**

```
    /* Ams_traf_gen expects two interrupts:                    */
    /* 1. A self interrupt that signals a new call.           */
    /* 2. A spurious interrupt.                               */

5   /* If this is not a call start handle the spurious interrupt;  */
    /* otherwise, the transition conditions will cause the process to */
    /* go to the 'start' state.                               */
    if (! CALL_START)
        {
10  /* This is a spurious interrupt. */
        badd_call_req_spurious_signal_handle ();
        }
```

transition **wait_call_idle -> wait_call_idle**

attribute	*value*	*type*	*default value*
name	tr_4	string	tr
condition	default	string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

transition **wait_call_idle -> start**

attribute	*value*	*type*	*default value*
name	tr_102	string	tr
condition	CALL_START	string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

...
...

forced state **start**

attribute	*value*	*type*	*default value*
name	start	string	st
enter execs	(See below.)	textlist	
exit execs	(empty)	textlist	(empty)
status	forced	toggle	unforced

enter execs **start**

```
   /* Start a call by issuing a call open request to the ATM                    */
   /* adaptation layer.                                                         */
   if (LTRACE_ACTIVE)
       {
 5     op_prg_odb_print_major (pid_string, "Initiating call Request.",
                   "Sending Request signal to Call Scheduler.", OPC_NIL);
       }


10 /* Code added by Clark */
   if (LTRACE_CALL_REQUESTOR_ACTIVE)
   {
     printf("In badd_call_requestor, start; reached start state.\n");
   } /* if (LTRACE_CALL_REQUESTOR_ACTIVE) */
15
   /* Create and set the fields in the interface ICI.                  */
   if_iciptr = op_ici_create ("badd_call_req_if_ici");

   /* Code commented out for badd testing */
20 op_ici_install (if_iciptr);
   /* End commented out code */

   badd_packet_size = packet_size;

25 /* Compute the peak cell rate in cells/second, which is set to 1          */
   /* times the average cell rate in this example.                          */
   peak_cell_rate = (1.0 / int_arr_time) * (badd_packet_size / AMSC_ATM_CELL_DATA_SIZE);

   op_ici_attr_set (if_iciptr, "interarrival time", int_arr_time);
30 op_ici_attr_set (if_iciptr, "packet size", packet_size);
   op_ici_attr_set (if_iciptr, "call wait time", call_wait_time);
   op_ici_attr_set (if_iciptr, "call duration", call_duration);
   op_ici_attr_set (if_iciptr, "dest addr", dest_addr);
   op_ici_attr_set (if_iciptr, "QoS class", qos_class);
35 op_ici_attr_set (if_iciptr, "AAL type", AAL_type);
   op_ici_attr_set (if_iciptr, "peak cell rate", peak_cell_rate);

   /* Code added by Clark */
   if (LTRACE_CALL_REQUESTOR_ACTIVE)
40 {
     printf("In badd_call_requestor, start; starting call to dest %d.\n",
         dest_addr);
   } /* if (LTRACE_CALL_REQUESTOR_ACTIVE) */

45 /* End of code added by Clark */

   /* Send a remote interrupt which will carry the ICI to the AAL              */
```

```
/* module.                                          */

/*  Code commented out for badd testing */
op_intrpt_schedule_remote (op_sim_time (), BADD_REQUEST_SIGNAL, sch_module_id);
/*  End commented out code */

/*  Generate a self intrpt to wake up this process for generating  */
/*  another call request.                            */
if (LTRACE_CALL_REQUESTOR_ACTIVE)
{
  event_time = op_sim_time() + op_dist_outcome (call_duration_distptr);
  printf("In clark_badd_call_requestor.start; schedule call completion for %f.\n",
      event_time);
}

op_intrpt_schedule_self (op_sim_time () +
    op_dist_outcome (call_duration_distptr), AMSC_TGEN_WAIT_CALL_DURATION);
```

transition **start -> wait_call_duration**

attribute	*value*	*type*	*default value*
name	tr_105	string	tr
condition		string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

unforced state **wait_call_duration**

attribute	*value*	*type*	*default value*
name	wait_call_duration	string	st
enter execs	(empty)	textlist	(empty)
exit execs	(See below.)	textlist	
status	unforced	toggle	unforced

exit execs **wait_call_duration**

```
/* wait_call_duration expects two interrupts:                    */
/* 1. A self interrupt that signals expired call duration time.  */
/* 2. A spurious interrupt.                                      */

/* If this is not a call start handle the spurious interrupt;    */
/* otherwise, the transition conditions will cause the process to */
/* go to the 'start' state.                                      */
if (! WAIT_CALL_DURATION)
    {
  /* This is a spurious interrupt. */
    badd_call_req_spurious_signal_handle ();
    }
```

...
...

transition **wait_call_duration -> wait_call_duration**			
attribute	*value*	*type*	*default value*
name	tr_96	string	tr
condition	default	string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

transition **wait_call_duration -> schedule**			
attribute	*value*	*type*	*default value*
name	tr_100	string	tr
condition	WAIT_CALL_DURATION	string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

APPENDIX D. BADD_CALL_GENERATOR.

...
...

External File Set

attribute	*value*	*type*	*default value*
external file set	badd_functions	typed file	

Process Model Comments

General Process Description:

The ams_traf_gen process initiates call start and end, generates the actual packets, and sends them to the AAL module. It provides an example of an AAL client process.

ICI Interfaces:

---

ams_if_ici

Packet Formats:

None.

Statistic Interfaces:

None

Process Registry:

Published Attributes: None
Expected Attributes: None

Restrictions:

None

Process Model Interface Attributes

Attribute **begsim intrpt** properties

Property	*Value*	*Inherit*
Assign Status:	hidden	
Initial Value:	enabled	N/A
Default Value:	disabled	YES
Data Type:	toggle	N/A
Attribute Description:	Private	N/A
Comments:	This attribute specifies whether a 'begin simulation interrupt' is generated for a processor module's root process at the start of the simulation.	YES
Symbol Map:	NONE	YES

Attribute **endsim intrpt** properties

Property	*Value*	*Inherit*
Assign Status:	hidden	

Initial Value:	disabled	N/A
Default Value:	disabled	YES
Data Type:	toggle	N/A
Attribute Description:	Private	N/A
Comments:		YES
	This attribute specifies whether an 'end simulation interrupt' is generated for a processor module's root process at the end of the simulation.	
Symbol Map:	NONE	YES

Attribute **failure intrpts** properties

Property	*Value*	*Inherit*
Assign Status:	hidden	
Initial Value:	disabled	N/A
Default Value:	disabled	YES
Data Type:	enumerated	N/A
Attribute Description:	Private	N/A
Comments:		YES
	This attribute specifies whether failure interrupts are generated for a processor module's root process upon failure of nodes or links in the network model.	
Symbol Map:	NONE	YES

Attribute **intrpt interval** properties

Property	*Value*	*Inherit*
Assign Status:	hidden	
Initial Value:	disabled	N/A
Default Value:	disabled	YES
Data Type:	toggle double	N/A
Attribute Description:	Private	N/A
Units:	sec.	YES
Comments:		YES
	This attribute specifies how often regular interrupts are scheduled for the root process of a processor module.	
Symbol Map:	NONE	YES

Attribute **priority** properties

Property	*Value*	*Inherit*
Assign Status:	hidden	
Initial Value:	0	N/A
Default Value:	0	YES
Data Type:	integer	N/A
Attribute Description:	Private	N/A
Low Range:	-32767 inclusive	YES
High Range:	32767 inclusive	YES
Comments:		YES
	This attribute is used to determine the execution order of events that are scheduled to occur at the same simulation time.	
Symbol Map:	NONE	YES

...
...

Attribute **recovery intrpts** properties		
Property	*Value*	*Inherit*
Assign Status:	hidden	
Initial Value:	disabled	N/A
Default Value:	disabled	YES
Data Type:	enumerated	N/A
Attribute Description:	Private	N/A
Comments:	This attribute specifies whether recovery interrupts are scheduled for the processor module's root process upon recovery of nodes or links in the network model.	YES
Symbol Map:	NONE	YES

Attribute **super priority** properties		
Property	*Value*	*Inherit*
Assign Status:	hidden	
Initial Value:	disabled	N/A
Default Value:	disabled	YES
Data Type:	toggle	N/A
Attribute Description:	Private	N/A
Comments:	This attribute is used to determine the execution order of events that are scheduled to occur at the same simulation time.	YES
Symbol Map:	NONE	YES

Process Model Simulation Attributes

Attribute **class_A_CDV_tolerance** properties	
Property	*Value*
Default Value:	0.0
Data Type:	double
Attribute Description:	Private
Auto. assign value:	FALSE
Comments:	Upper bound on cell delay variance that two consecutive Quality of Service (QoS) class A cells may experience.

Attribute **class_B_CDV_tolerance** properties	
Property	*Value*
Default Value:	0.0
Data Type:	double
Attribute Description:	Private
Auto. assign value:	FALSE
Comments:	Upper bound on cell delay variance that two

	consecutive Quality of Service (QoS) class B cells may experience.

Attribute **class_C_CDV_tolerance** properties

Property	*Value*
Default Value:	0.0
Data Type:	double
Attribute Description:	Private
Auto. assign value:	FALSE
Comments:	Upper bound on cell delay variance that two consecutive Quality of Service (QoS) class C cells may experience.

Attribute **class_D_CDV_tolerance** properties

Property	*Value*
Default Value:	0.0
Data Type:	double
Attribute Description:	Private
Auto. assign value:	FALSE
Comments:	Upper bound on cell delay variance that two consecutive Quality of Service (QoS) class D cells may experience.

Header Block

```
#include "/usr/local/mil3_dir/3.0.A_DL5/models/std/atm/ams_interfaces.h"
#include "/usr/local/mil3_dir/3.0.A_DL5/models/std/atm/ams_aal_interfaces.h"
/* Code added by Clark */
#include "/usr/work/benton/op_code/badd_interface.h"
/* end code added by Clark */

/* These are transition conditions. */
#define SIGNAL                  ((op_intrpt_type () == OPC_INTRPT_REMOTE) &&     \
                                 (op_intrpt_code () == AMSC_INTERFACE_SIGNAL))

#define EST_CON                 (primitive == AMSC_AAL_ESTAB_Con)

#define REL_IND                 (primitive == AMSC_AAL_RELEASE_Ind)

#define REL_CON                 (primitive == AMSC_AAL_RELEASE_Con)

#define CALL_END                ((op_intrpt_type () == OPC_INTRPT_SELF) &&       \
                                 (op_intrpt_code () == AMSC_TGEN_CALL_END))

/* These are the ams_traf_gen self interrupt codes. */
#define   AMSC_TGEN_CALL_START   0
#define AMSC_TGEN_CALL_END   1
```

```
#define AMSC_TGEN_DATA_GEN  2
#define BADD_CALL_RESCHEDULE  5
#define BADD_CHECK_CHANNEL  6

/* The ams_traf_gen process will output trace information if this conditional is true. */
#define LTRACE_ACTIVE (debug_mode && .                                    \
                       (op_prg_odb_ltrace_active ("ams") ||               \
                        op_prg_odb_ltrace_active ("ams_traf") ||          \
                        op_prg_odb_ltrace_active ("ams_traf_gen")))

#define LTRACE_CONNECT_ACTIVE            (op_prg_odb_ltrace_active ("connect"))

/* Code added for badd */
#define LTRACE_STATIC_PCR_ACTIVE         (op_prg_odb_ltrace_active ("static_pcr"))

#define LTRACE_CALL_GENERATOR_ACTIVE     (op_prg_odb_ltrace_active ("call_generator"))

#define LTRACE_CALL_GEN_ACTIVE           (op_prg_odb_ltrace_active ("call_gen"))

#define LTRACE_CALL_RESCHEDULER_ACTIVE   (op_prg_odb_ltrace_active ("call_rescheduler"))

#define LTRACE_CALL_COMPLETE_ACTIVE      (op_prg_odb_ltrace_active ("call_complete"))

/* Procedure declarations. */
void badd_call_gen_spurious_signal_handle ();
void badd_call_gen_error ();
```

State Variable Block

```
int             \my_pro_id;
int             \dest_addr;
int             \packet_size;
int             \AAL_type;
int             \qos_class;
int             \channel_assigned;
int             \debug_mode;
int             \to_aal_stream_index;
double          \int_arr_time;
double          \call_wait_time;
double          \call_duration;
double          \peak_cell_rate;
double          \avg_rate;
double          \channel_delay;
char            \pid_string [128];
Objid           \my_id;
Objid           \aal_module_id;
Objid           \parent_obj_id;
Ici*            \aal_handle_iciptr;
Ici*            \channel_iciptr;
```

```
Evhandle                \next_packet_arrival;
Evhandle                \call_end_intrpt;
Distribution*       \int_arrival_distptr;
Distribution*       \call_duration_distptr;
Prohandle                   \my_prohandle;
Badd_Sch_Mod_Data*      \module_data_ptr;
AmsT_Traf_Contract*     \traf_con_ptr;
AmsT_Neighbor_Data*     \nbr_data_ptr;
```

Temporary Variable Block

```
/* Code added for BADD */
int                     primitive;
int                     badd_dest_addr;
double          start_time;
double          badd_packet_size;
double                  event_time;
double                  end_sim_time;
double                  cdv_tolerance;
extern int              total_calls_generated;
extern int              total_calls_received;
Ici*                    call_iciptr;
Ici*            upper_handle_iciptr;
Ici*                    if_iciptr;
Packet*                 pkptr;
AmsT_Traf_Contract*     tmp_traf_con_ptr;

/* End code added for BADD */
```

Function Block

```
void
badd_call_gen_spurious_signal_handle ()
    {
    Ici*                if_iciptr;
    Ici*                release_if_iciptr;
    int                 primitive;
    Ici*                ll_handle_iciptr;
    Packet*             pkptr;

    /* This procedure handles spurious interrupts.                  */
    /* Only three types of spurious interrupts can be accepted. All */
    /* others must result in an op_sim_end (). These three          */
    /* interrupts are:                                              */
    /*   1. An AAL ESTAB Ind.                                       */
    /*   2. An AAL RELEASE Con.                                     */
    /*   3. A packet arrival.                                       */
    FIN (ams_traf_gen_spurious_signal_handle ());

    if (SIGNAL)
```

```
{
/* This is a signal from the AAL.                                    */

/* Obtain the ICI pointer.                                          */
if_iciptr = op_intrpt_ici ();

/* Obtain the primitive from the ICI.                               */
op_ici_attr_get (if_iciptr, "primitive", &primitive);

/* Obtain the 'lower layer handle' from the ICI.                    */
op_ici_attr_get (if_iciptr, "lower layer handle", &ll_handle_iciptr);

/* Switch on the 'primitive'.          */
switch (primitive)
    {
    case AMSC_AAL_ESTAB_Ind:
        {
        /* This is an 'establish indication' from the AAL.          */

        if (LTRACE_ACTIVE)
            {
            op_prg_odb_print_major (pid_string, "Received spurious AAL ESTABLISH Request signal.",
                "Call not accepted.", "Sending AAL RELEASE Request signal.", OPC_NIL);
            }

        /* Set the 'upper layer handle' in the interface            */
        /* ICI for the 'establish indication' signal, since         */
        /* the lower layer expects it to be filled in. The          */
        /* ICI is not destroyed, since this is a forced             */
        /* interrupt and the lower layer process expects            */
        /* to obtain the handle from the ICI when the               */
        /* control returns.                                         */
        op_ici_attr_set (if_iciptr, "upper layer handle", OPC_NIL);

        /* Simply send an AAL_RELEASE_Req to request that           */
        /* the connection be released.                              */
        release_if_iciptr = op_ici_create (AMSC_INTERFACE_ICI);
        op_ici_attr_set (release_if_iciptr, "primitive", AMSC_AAL_RELEASE_Req);
        op_ici_attr_set (release_if_iciptr, "lower layer handle", ll_handle_iciptr);
      op_ici_install (release_if_iciptr);

        /* Send a remote interrupt which will carry the ICI to the AAL module. */
        op_intrpt_schedule_remote (op_sim_time (), AMSC_INTERFACE_SIGNAL, aal_module_id);

        break;
        }

    case AMSC_AAL_RELEASE_Con:
        {
        /* This is a 'release confirm' from the AAL.                */

        if (LTRACE_ACTIVE)
            {
            op_prg_odb_print_major (pid_string, "Received spurious AAL RELEASE Confirm signal.",
                "This is in response to AAL RELEASE Request terminating spurious connection.
            }

        /* This is a response to our earlier 'release               */
        /* request' which was a response to the                     */
```

```
                    /* spurious 'estab indication'.                          */
                    /* Do nothing other than destroy the ICI.                */
                    op_ici_destroy (if_iciptr);

                    break;
                    }

                default:
                    {
                    /* This is some other completely unexpected              */
                    /* signal. Issue error message and terminate             */
                    /* simulation.                                           */
                    op_sim_end ("Received unexpected signal.", "", "", "");
                    }
                }
            }
        else if (op_intrpt_type () == OPC_INTRPT_STRM)
            {
            /* This is a spurious packet arrival. The ams_traf_gen           */
            /* just destroys this packet.                                    */
            if (LTRACE_ACTIVE)
                {
                op_prg_odb_print_major (pid_string, "Received spurious DATA packet.",
                    "Destroying the packet.", OPC_NIL);
                }

            /* Get the packet fromt the stream and destroy it.               */
            pkptr = op_pk_get (op_intrpt_strm ());
            op_pk_destroy (pkptr);
            }
        else
            {
            /* This is some other completely unexpected                      */
            /* interrupt. Issue error message and terminate                  */
            /* simulation.                                                   */
            op_sim_end ("Received unexpected interrupt.", "", "", "");
            }

        FOUT;
        }

void
badd_call_gen_error (msg0, msg1, msg2)
        char*           msg0;
        char*           msg1;
        char*           msg2;
        {
        /** Print an error message and exit the simulation. **/
        FIN (badd_call_gen_error (msg0, msg1, msg2));

        op_sim_end ("Error in badd_call_generator process:",
            msg0, msg1, msg2);

        FOUT;
        }
```

Diagnostic Block

```
/* Display connection information, if requested. */
if (LTRACE_CONNECT_ACTIVE)
    {
    /* Print information. */
    ams_neighbor_data_print (nbr_data_ptr, ams_neighbor_desc_print_noop);
    }
```

forced state **INIT**

attribute	*value*	*type*	*default value*
name	INIT	string	st
enter execs	(See below.)	textlist	
exit execs	(empty)	textlist	(empty)
status	forced	toggle	unforced

enter execs **INIT**

```
/* New process created for BADD ATM Simulation.                */
/* This process generates the call data when the call is scheduled */
/* for a channel.                                              */

/* Determine if the simulation is in debug mode, traces are only  */
/* enabled in the debug mode.                                  */
debug_mode = op_sim_debug();

/* Determine the prohandle and process_id for this instances of a */
/* badd_call_generator.                                        */
my_prohandle = op_pro_self();
my_pro_id = op_pro_id(my_prohandle);

if (my_pro_id == OPC_PRO_ID_INVALID)
  badd_call_gen_error("Unable to get own process id", OPC_NIL, OPC_NIL);

parent_obj_id = op_pro_mod_objid(my_prohandle);

/* Initialize the process ID display string                    */
sprintf(pid_string, "badd_call_gen PID (%d)", my_pro_id);

/* Access the module memory data structure                     */
module_data_ptr = (Badd_Sch_Mod_Data*)op_pro_modmem_access();
if (module_data_ptr == OPC_NIL)
  badd_call_gen_error("Unable to get module memory.", OPC_NIL, OPC_NIL);

nbr_data_ptr = module_data_ptr->md_neighbor_data_ptr;
channel_delay = module_data_ptr->md_channel_delay;

/* Access the argument memory and get the call descriptor      */
/* information.                                                */
call_iciptr = (Ici*)op_pro_argmem_access();

if (LTRACE_CALL_GENERATOR_ACTIVE)
{
```

```
    printf("In badd_call_gen.init: my_prohandle = %x.\n", my_prohandle);
    printf("In badd_call_gen.init: call_iciptr = %x.\n", call_iciptr);
    printf("In badd_call_gen.init; pro_id = %d.\n", my_pro_id);
  }

  if (call_iciptr == OPC_NIL)
    badd_call_gen_error("In badd_call_gen.init: unable to get call_iciptr.", OPC_NIL, OPC_NIL);

  if ((op_ici_attr_get (call_iciptr, "interarrival time", &int_arr_time) == OPC_COMPCODE_FAILURE) ||
    (op_ici_attr_get (call_iciptr, "packet size", &packet_size) == OPC_COMPCODE_FAILURE)      ||
    (op_ici_attr_get (call_iciptr, "call wait time", &call_wait_time) == OPC_COMPCODE_FAILURE) ||
    (op_ici_attr_get (call_iciptr, "call duration", &call_duration) == OPC_COMPCODE_FAILURE)   ||
    (op_ici_attr_get (call_iciptr, "dest addr", &dest_addr) == OPC_COMPCODE_FAILURE)          ||
    (op_ici_attr_get (call_iciptr, "QoS class", &qos_class) == OPC_COMPCODE_FAILURE)          ||
    (op_ici_attr_get (call_iciptr, "AAL type", &AAL_type) == OPC_COMPCODE_FAILURE)            ||
    (op_ici_attr_get (call_iciptr, "channel assigned", &channel_assigned) == OPC_COMPCODE_FAILURE)          ||
    (op_ici_attr_get (call_iciptr, "peak cell rate", &peak_cell_rate) == OPC_COMPCODE_FAILURE))
    badd_call_gen_error("In start call, unable to get values from call_iciptr", OPC_NIL, OPC_NIL);

  if (LTRACE_CALL_GENERATOR_ACTIVE)
  {
    printf("In badd_call_gen.init; packet_size = %d.\n", packet_size);
    printf("In badd_call_gen.init; dest_addr = %d.\n", dest_addr);
    printf("In badd_call_gen.init; AAL_type = %d.\n", AAL_type);
    printf("In badd_call_gen.init; qos_class = %d.\n", qos_class);
    printf("In badd_call_gen.init; int_arr_time = %f.\n", int_arr_time);
    printf("In badd_call_gen.init; call_wait_time = %f.\n", call_wait_time);
    printf("In badd_call_gen.init; call_duration = %f.\n", call_duration);
    printf("In badd_call_gen.init; channel assigned = %d.\n", channel_assigned);
  }

  /* Release the memory for this call descriptor */
  op_ici_destroy(call_iciptr);

  /* Load in the packet interarrival, call wait and call duration                */
  /* distributions.                                                              */
  int_arrival_distptr  = op_dist_load ("constant", int_arr_time, 0.0);
  call_duration_distptr = op_dist_load ("constant", call_duration, 0.0);

  /* Code added for BADD */
  badd_packet_size = packet_size;
  peak_cell_rate = (1.0 / int_arr_time) * (badd_packet_size / AMSC_ATM_CELL_DATA_SIZE);

  if (LTRACE_CALL_GENERATOR_ACTIVE)
  {
    printf("In badd_call_gen, Init; Peak_Cell_Rate = %f.\n", peak_cell_rate);
    printf("\tInt_Arr_Time = %f, \tPacket_Size = %d.\n", int_arr_time, packet_size);
  }

  /* Code for getting the simulation duration */
  op_ima_sim_attr_get(OPC_IMA_DOUBLE, "duration", &end_sim_time);

  if (LTRACE_CALL_GENERATOR_ACTIVE)
  {
    printf("In badd_call_gen, Init; end_sim_time = %f.\n", end_sim_time);
  }
  /* End Code added for BADD */

  /* Use the default Cell Delay Variation (CDV) tolerance for the                */
```

```
95  /* specific QoS class.                                          */
    cdv_tolerance = ams_CDV_tolerance_default_obtain (qos_class);

    /* Create a traffic contract for calls originating from this src.  */
    /* There is no return traffic, but allocate a bit (.1 * PCR) of    */
100 /* bandwidth anyway.                                            */
    traf_con_ptr = ams_traffic_contract_create (peak_cell_rate,
        peak_cell_rate * 0.1, cdv_tolerance, cdv_tolerance);

    /* Initialize the AAL module object ID and aal_stream_index.  */
105 aal_module_id = module_data_ptr->md_aal_module_id;
    to_aal_stream_index = module_data_ptr->md_to_aal_stream_index;

    if (LTRACE_CALL_GENERATOR_ACTIVE)
      printf("In badd_call_gen; aal_module_id = %d.\n", aal_module_id);
110
    /* Generate PID display string.                                 */
    sprintf (pid_string, "badd_call_gen PID (%d)", op_pro_id (op_pro_self ()));
```

transition **INIT -> start**

attribute	*value*	*type*	*default value*
name	tr_98	string	tr
condition		string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

forced state **start**

attribute	*value*	*type*	*default value*
name	start	string	st
enter execs	(See below.)	textlist	
exit execs	(empty)	textlist	(empty)
status	forced	toggle	unforced

enter execs **start**

```
    /* Start a call by issuing a call open request to the ATM       */
    /* adaptation layer.                                            */
    if (LTRACE_ACTIVE)
        {
5       op_prg_odb_print_major (pid_string, "Initiating call SETUP.",
                    "Sending AAL ESTABLISH Request signal.", OPC_NIL);
        }


10  /* Make a copy of the traffic contract, since the lower layers  */
    /* are responsible for destroy the data structure.              */
    tmp_traf_con_ptr = ams_traffic_contract_copy (traf_con_ptr);

    upper_handle_iciptr = op_ici_create("badd_call_gen_handle");
```

```
15  op_ici_attr_set (upper_handle_iciptr, "call_gen_prohandle", &my_prohandle);


    /* Create and set the fields in the interface ICI.                    */
    if_iciptr = op_ici_create (BADD_CALL_GEN_IF_ICI);
20  op_ici_install (if_iciptr);
    op_ici_attr_set (if_iciptr, "primitive", AMSC_AAL_ESTAB_Req);
    op_ici_attr_set (if_iciptr, "address", dest_addr);
    op_ici_attr_set (if_iciptr, "called party SAP", AMSC_AAL_SAP_ANY);
    op_ici_attr_set (if_iciptr, "QoS class", qos_class);
25  op_ici_attr_set (if_iciptr, "upper layer handle", upper_handle_iciptr);
    op_ici_attr_set (if_iciptr, "AAL type", AAL_type);
    op_ici_attr_set (if_iciptr, "traffic contract", tmp_traf_con_ptr);
    op_ici_attr_set (if_iciptr, "badd_call_gen_pro_id", my_pro_id);
    op_ici_attr_set (if_iciptr, "badd_call_gen_channel_id", channel_assigned);
30
    /* Code added for BADD */

    if (LTRACE_CALL_GENERATOR_ACTIVE)
    {
35    printf("In badd_call_generator, start; starting call to dest %d.\n", dest_addr);
      printf("In badd_call_generator, start; aal_module_id = %d.\n", aal_module_id);
    } /* End if (LTRACE_CALL_GENERATOR_ACTIVE) */

    /* End of code added for BADD */
40
    /* Send a remote interrupt which will carry the ICI to the AAL        */
    /* module.                                                            */

    op_intrpt_schedule_remote (op_sim_time (), AMSC_INTERFACE_SIGNAL, aal_module_id);
```

transition **start -> EstCon**

attribute	*value*	*type*	*default value*
name	tr_105	string	tr
condition		string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

unforced state **EstCon**

attribute	*value*	*type*	*default value*
name	EstCon	string	st
enter execs	(See below.)	textlist	
exit execs	(See below.)	textlist	
status	unforced	toggle	unforced

enter execs **EstCon**

```
if (LTRACE_CALL_GENERATOR_ACTIVE)
{
```

```
    printf("In call_gen.EstCon.enter execs.\n", OPC_NIL, OPC_NIL);
}
```

exit execs **EstCon**

```
/* This state expects three possible interrupts:                    */
/* 1. Establish Confirm signal.                                     */
/* 2. Release Indication signal.                                    */
/* 3. Spurious signal.                                              */

if (LTRACE_CALL_GENERATOR_ACTIVE)
{
   printf("In badd_call_gen.EstCon: call_gen restarted.\n");
}

/* Determine what signal arrived.                                   */
/* Obtain the interface ICI pointer.                                */
if_iciptr = op_intrpt_ici ();

/* Obtain the primitive value.                                      */
op_ici_attr_get (if_iciptr, "primitive", &primitive);

/* Switch off the primitive value.                                  */
    switch (primitive)
        {
        case AMSC_AAL_ESTAB_Con:
            {
            /* The signal is an AAL ESTABLISH Confirm signal.       */
            /* The connection has been established.                 */
            if (LTRACE_ACTIVE)
            op_prg_odb_print_major (pid_string, "Received AAL ESTABLISH Confirm signal.",
                                    "Connection established.", OPC_NIL);

            /* Obtain the lower layer handle.                       */
            op_ici_attr_get (if_iciptr, "lower layer handle", &aal_handle_iciptr);

            /* Destroy the ICI.                                     */
            op_ici_destroy (if_iciptr);

            /* Schedule the call end event.                         */
            call_end_intrpt = op_intrpt_schedule_self (op_sim_time () +
                    op_dist_outcome (call_duration_distptr), AMSC_TGEN_CALL_END);

            /* Schedule the next arrival.                           */
            next_packet_arrival = op_intrpt_schedule_self (op_sim_time () +
                    op_dist_outcome (int_arrival_distptr), AMSC_TGEN_DATA_GEN);

        if (LTRACE_CALL_GENERATOR_ACTIVE)
        {
          start_time = op_sim_time () + op_dist_outcome (int_arrival_distptr);
          printf("In badd_call_gen.EstCon: start time = %f.\n", start_time);
        }

            break;
            }

          case AMSC_AAL_RELEASE_Ind:
            {
            /* The signal is an AAL RELEASE Indication signal.      */
```

```
55          /* The connection has been terminated.                    */
            if (LTRACE_ACTIVE)
                op_prg_odb_print_major (pid_string, "Received AAL RELEASE Indication signal.",
                        "Connection terminated.", OPC_NIL);

60      /* Destroy the ICI.                                           */
            op_ici_destroy (if_iciptr);
            break;
            }

65      default:
            {
            /* This is a spurious signal.                 */
        printf("In badd_call_gen.EstCon state; inside switch statement.\n");
            badd_call_gen_spurious_signal_handle ();
70          break;
            }
    }
```

transition **EstCon -> data gen**			
attribute	*value*	*type*	*default value*
name	tr_10	string	tr
condition	EST_CON	string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

transition **EstCon -> reschedule**			
attribute	*value*	*type*	*default value*
name	tr_100	string	tr
condition	REL_IND	string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

transition **EstCon -> EstCon**			
attribute	*value*	*type*	*default value*
name	tr_110	string	tr
condition	default	string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

unforced state **data gen**			
attribute	*value*	*type*	*default value*
name	data gen	string	st

enter execs	(See below.)	textlist	
exit execs	(See below.)	textlist	
status	unforced	toggle	unforced

enter execs **data gen**

```
if (LTRACE_CALL_GENERATOR_ACTIVE)
{
  printf("In call_gen, received signal to start sending data.\n", OPC_NIL, OPC_NIL);
}
```

exit execs **data gen**

```
/* This state expects four possible interrupts:                    */
/* 1. An AAL RELEASE Indication signal.                            */
/* 2. A Spurious signal.                                           */
/* 3. A call end interrupt.                                        */
/* 4. A generate data interrupt.                                   */

if (LTRACE_CALL_GENERATOR_ACTIVE)
{
  printf("In badd_call_gen.data gen.exit execs.\n");
}



/* Determine if this is an AAL signal.                             */
if (SIGNAL)
    {
    /* Obtain the interface ICI and enclosed primitive.            */
    if_iciptr = op_intrpt_ici ();
    op_ici_attr_get (if_iciptr, "primitive", &primitive);

    /* If the primitive is 'release indication',                   */
    /* cancel the call end and data gen intrpts;                   */
    /* otherwise, the primitive indicates a                        */
    /* spurious signal.                                            */
    if (primitive == AMSC_AAL_RELEASE_Ind)
        {
        /* This is a 'release indication' signal.                  */
        if (LTRACE_ACTIVE)
            op_prg_odb_print_major (pid_string, "Received AAL RELEASE Indication signal.",
                    "Connection terminated.", OPC_NIL);

        /* Cancel the 'call end' and 'generate data' events.       */
        op_ev_cancel (next_packet_arrival);
        op_ev_cancel (call_end_intrpt);

        /* Destroy the ICI.                                        */
        op_ici_destroy (if_iciptr);
        }
    else
        {
        /* This is a spurious signal.                              */
        ams_traf_gen_spurious_signal_handle ();
        }
    }
```

```
45
   /* Determine whether this interrupt is a self interrupt                    */
   /* that indicates the end-of-call.                                         */
   else if (CALL_END)
        {
50      /* This is a self interrupt that indicates the end-of-call.           */
        if (LTRACE_ACTIVE)
             {
             op_prg_odb_print_major (pid_string, "Initiating call END.",
                       "Sending AAL RELEASE Request signal.", OPC_NIL);
55           }

        /* Cancel the current packet arrival interrupt.                       */
        op_ev_cancel (next_packet_arrival);

60      /* Notify the AAL that the call is complete.                          */
        if_iciptr = op_ici_create (BADD_CALL_GEN_IF_ICI);
        op_ici_attr_set (if_iciptr, "primitive", AMSC_AAL_RELEASE_Req);
        op_ici_attr_set (if_iciptr, "lower layer handle", aal_handle_iciptr);
        op_ici_attr_set (if_iciptr, "badd_call_gen_channel_id", channel_assigned);
65    op_ici_install (if_iciptr);

     if (LTRACE_CALL_GENERATOR_ACTIVE)
     {
       printf("In call_gen.data_gen: call completed for channel %d.\n", channel_assigned);
70   }

     /* Send a remote interrupt that  will carry the ICI to the AAL           */
       /* module.                                                             */
       op_intrpt_schedule_remote (op_sim_time (), AMSC_INTERFACE_SIGNAL, aal_module_id);
75

        }
   else if (op_intrpt_type () == OPC_INTRPT_STRM)
        {
        /* This is a spurious signal.                                         */
80      badd_call_gen_spurious_signal_handle ();
        }
   else
        {
        /* This is a 'generate data' event.                                   */
85      if (LTRACE_ACTIVE)
             op_prg_odb_print_major (pid_string, "Sending DATA.", OPC_NIL);

        /* Install the AAL handle ICI.                                        */
        op_ici_install (aal_handle_iciptr);
90
        /* Create a data packet and send it to the AAL.                       */
        pkptr = op_pk_create (packet_size);

     if (LTRACE_CALL_GENERATOR_ACTIVE)
95   {
       printf("In badd_call_gen.data gen; pk created with addr %d.\n", dest_addr);
       printf("In badd_call_gen.data gen; pk created with size = %d.\n", packet_size);
       printf("In badd_call_gen.data gen; pk send to strm index %d. \n", to_aal_stream_index);
     }
100
        op_pk_send (pkptr, to_aal_stream_index);

        /* Schedule the next arrival.                                         */
```

...
...

```
105     next_packet_arrival = op_intrpt_schedule_self (op_sim_time () +
                op_dist_outcome (int_arrival_distptr), AMSC_TGEN_DATA_GEN);

    }
```

transition **data gen -> RelCon**

attribute	*value*	*type*	*default value*
name	tr_14	string	tr
condition	CALL_END	string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

transition **data gen -> reschedule**

attribute	*value*	*type*	*default value*
name	tr_101	string	tr
condition	REL_IND	string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

transition **data gen -> data gen**

attribute	*value*	*type*	*default value*
name	tr_111	string	tr
condition	default	string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

unforced state **RelCon**

attribute	*value*	*type*	*default value*
name	RelCon	string	st
enter execs	(See below.)	textlist	
exit execs	(See below.)	textlist	
status	unforced	toggle	unforced

enter execs **RelCon**

```
if (LTRACE_CALL_GENERATOR_ACTIVE)
{
  printf("In call_gen.RelCon.enter execs.\n", OPC_NIL, OPC_NIL);
}
```

exit execs **RelCon**

```
   /* This state expects two possible interupts:                    */
   /* 1. An AAL RELEASE Confirm signal.                             */
   /* 2. A spurious signal.                                         */

   /* Determine what signal arrived.                                */
   /* Obtain the interface ICI and enclosed primitive.              */

   if_iciptr = op_intrpt_ici ();

   op_ici_attr_get (if_iciptr, "primitive", &primitive);

   /* If this is a 'AAL RELEASE Confirm' signal, do nothing;        */
   /* otherwise, this is a spurious signal.                         */
   if (primitive == AMSC_AAL_RELEASE_Con)
   {

     /*printf("In call-generator.RelCon: Peak Cell Rate = %f.\n", peak_cell_rate); */
     /* This is a 'AAL RELEASE Confirm' signal.                     */
     if (LTRACE_CALL_GENERATOR_ACTIVE)
     {
       op_prg_odb_print_major (pid_string, "Received RELEASE Confirm signal.",
             "Call END complete.", "Connection terminated.", OPC_NIL);

       /* Destroy the ICI.                                                */
       op_ici_destroy (if_iciptr);
     }
   }
   else
   {
     /* This is a spurious signal.                                  */
     badd_call_gen_spurious_signal_handle ();
   }
```

transition **RelCon -> call done**

attribute	*value*	*type*	*default value*
name	tr_103	string	tr
condition	REL_CON	string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

transition **RelCon -> RelCon**

attribute	*value*	*type*	*default value*
name	tr_113	string	tr
condition	default	string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

...
...

unforced state **reschedule**

attribute	*value*	*type*	*default value*
name	reschedule	string	st
enter execs	(See below.)	textlist	
exit execs	(empty)	textlist	(empty)
status	unforced	toggle	unforced

enter execs **reschedule**

```
if (LTRACE_CALL_GENERATOR_ACTIVE)
{
  printf("In call_gen.reschedule.enter execs.\n", OPC_NIL, OPC_NIL);
}

if_iciptr = op_ici_create ("badd_call_req_if_ici");
op_ici_install(if_iciptr);

op_ici_attr_set (if_iciptr, "interarrival time", int_arr_time);
op_ici_attr_set (if_iciptr, "packet size", packet_size);
op_ici_attr_set (if_iciptr, "call wait time", call_wait_time);
op_ici_attr_set (if_iciptr, "call duration", call_duration);
op_ici_attr_set (if_iciptr, "dest addr", dest_addr);
op_ici_attr_set (if_iciptr, "QoS class", qos_class);
op_ici_attr_set (if_iciptr, "AAL type", AAL_type);
op_ici_attr_set (if_iciptr, "peak cell rate", peak_cell_rate);
op_ici_attr_set (if_iciptr, "channel assigned", channel_assigned);

op_intrpt_schedule_remote(op_sim_time() + channel_delay, BADD_CALL_RESCHEDULE, parent_obj_id);

/* Need to send intrpt to run a reschedule event */
if (LTRACE_CALL_RESCHEDULER_ACTIVE)
{
  op_prg_odb_print_major(pid_string, "Call Terminated, Call Rescheduled.",
            "Destroying call_gen process.", OPC_NIL);
}


/* Send intrpt to check for next call on this channel */
channel_iciptr = op_ici_create("badd_channel_ici");
op_ici_install(channel_iciptr);
op_ici_attr_set (channel_iciptr, "channel number", channel_assigned);
if (LTRACE_CALL_RESCHEDULER_ACTIVE)
{
  printf("In call_gen.call_done, channel released = %d.\n", channel_assigned);
}
op_intrpt_schedule_remote(op_sim_time(), BADD_CHECK_CHANNEL, parent_obj_id);

/* This process destroys itself since the call is released */
if (op_pro_destroy(my_prohandle) == OPC_COMPCODE_FAILURE)
{
  badd_call_gen_error("Unable to terminate call_gen process.", OPC_NIL, OPC_NIL);
}
```

...

...

unforced state **call done**

attribute	*value*	*type*	*default value*
name	call done	string	st
enter execs	(See below.)	textlist	
exit execs	(empty)	textlist	(empty)
status	unforced	toggle	unforced

enter execs **call done**

```
   if (LTRACE_CALL_GENERATOR_ACTIVE)
   {
     op_prg_odb_print_major(pid_string, "Call Successfully Completed.",
                "Destroying call_gen process.", OPC_NIL);
 5 }

   /* Send intrpt to schedule next call */
   channel_iciptr = op_ici_create("badd_channel_ici");
   op_ici_install(channel_iciptr);
10 op_ici_attr_set (channel_iciptr, "channel number", channel_assigned);
   if (LTRACE_CALL_COMPLETE_ACTIVE)
   {
     printf("In call_gen.call_done, channel completed = %d.\n", channel_assigned);
   }
15 op_intrpt_schedule_remote(op_sim_time(), BADD_CHECK_CHANNEL, parent_obj_id);

   /* This process destroys itself since the call is completed */
   if (op_pro_destroy(my_prohandle) == OPC_COMPCODE_FAILURE)
   {
20   badd_call_gen_error("Unable to terminate call_gen process.", OPC_NIL, OPC_NIL);
   }
```

APPENDIX E.
BADD_CALL_SCHEDULER_NUMBER_BASED.

...
...

External File Set

attribute	*value*	*type*	*default value*
external file set	badd_functions	typed file	

Process Model Comments

General Process Description:

The badd_call_scheduler schedules and manages all static channel assignments for the BADD network. This model schedules calls by assigning new calls to the channel with the least number of calls current scheduled.

ICI Interfaces:

---

badd_call_req_if_ici
badd_call_gen_if_ici

Packet Formats:

None.

Statistic Interfaces:

None

Process Registry:

Published Attributes and Expected Attributes: None

Restrictions:

None

Process Model Attributes

Attribute **dest addr** properties

Property	*Value*	
Default Value:	NONE	
Data Type:	integer	
Attribute Description:	Private	
Auto. assign value:	FALSE	
Comments:	Specifies the destination of the call.	
Symbol Map:	*Symbol*	*Value*
	NONE	-1
Allow other values:	YES	

Attribute **scheduler delay** properties

Property	*Value*
Default Value:	0.0

Data Type:	double
Attribute Description:	Private
Auto. assign value:	FALSE
Units:	seconds
Comments:	The maximum time in seconds between scheduler events.

Attribute **transmission delay** properties

Property	*Value*
Default Value:	0.25
Data Type:	double
Attribute Description:	Private
Auto. assign value:	FALSE
Units:	seconds

Attribute **channel delay** properties

Property	*Value*
Default Value:	7.681E-05
Data Type:	double
Attribute Description:	Private
Auto. assign value:	FALSE
Units:	seconds
Comments:	The delay between calls on the same channel.

Attribute **calls pending** properties

Property	*Value*
Default Value:	0
Data Type:	integer
Attribute Description:	Private
Auto. assign value:	FALSE
Comments:	The maximum number of calls to store in the calls_pending list before running a schedule process.

Process Model Interface Attributes

Attribute **begsim intrpt** properties

Property	*Value*	*Inherit*
Assign Status:	hidden	
Initial Value:	enabled	N/A
Default Value:	disabled	YES
Data Type:	toggle	N/A
Attribute Description:	Private	N/A
Comments:	This attribute specifies whether a 'begin simulation interrupt' is generated for a processor module's root process at the start of the simulation.	YES

...
...

Symbol Map:	NONE	YES

Attribute **endsim intrpt** properties

Property	*Value*	*Inherit*
Assign Status:	hidden	
Initial Value:	enabled	N/A
Default Value:	disabled	YES
Data Type:	toggle	N/A
Attribute Description:	Private	N/A
Comments:		YES
	This attribute specifies whether an 'end simulation interrupt' is generated for a processor module's root process at the end of the simulation.	
Symbol Map:	NONE	YES

Attribute **failure intrpts** properties

Property	*Value*	*Inherit*
Assign Status:	hidden	
Initial Value:	disabled	N/A
Default Value:	disabled	YES
Data Type:	enumerated	N/A
Attribute Description:	Private	N/A
Comments:		YES
	This attribute specifies whether failure interrupts are generated for a processor module's root process upon failure of nodes or links in the network model.	
Symbol Map:	NONE	YES

Attribute **intrpt interval** properties

Property	*Value*	*Inherit*
Assign Status:	hidden	
Initial Value:	disabled	N/A
Default Value:	disabled	YES
Data Type:	toggle double	N/A
Attribute Description:	Private	N/A
Units:	sec.	YES
Comments:		YES
	This attribute specifies how often regular interrupts are scheduled for the root process of a processor module.	
Symbol Map:	NONE	YES

Attribute **priority** properties

Property	*Value*	*Inherit*
Assign Status:	hidden	
Initial Value:	0	N/A
Default Value:	0	YES
Data Type:	integer	N/A
Attribute Description:	Private	N/A
Low Range:	-32767 inclusive	YES
High Range:	32767 inclusive	YES

...
...

Comments:	This attribute is used to determine the execution order of events that are scheduled to occur at the same simulation time.	YES
Symbol Map:	NONE	YES

Attribute **recovery intrpts** properties

Property	*Value*	*Inherit*
Assign Status:	hidden	
Initial Value:	disabled	N/A
Default Value:	disabled	YES
Data Type:	enumerated	N/A
Attribute Description:	Private	N/A
Comments:	This attribute specifies whether recovery interrupts are scheduled for the processor module's root process upon recovery of nodes or links in the network model.	YES
Symbol Map:	NONE	YES

Attribute **super priority** properties

Property	*Value*	*Inherit*
Assign Status:	hidden	
Initial Value:	disabled	N/A
Default Value:	disabled	YES
Data Type:	toggle	N/A
Attribute Description:	Private	N/A
Comments:	This attribute is used to determine the execution order of events that are scheduled to occur at the same simulation time.	YES
Symbol Map:	NONE	YES

Process Model Simulation Attributes

Attribute **class_A_CDV_tolerance** properties

Property	*Value*
Default Value:	0.0
Data Type:	double
Attribute Description:	Private
Auto. assign value:	FALSE
Comments:	Upper bound on cell delay variance that two consecutive Quality of Service (QoS) class A cells may experience.

Attribute **class_B_CDV_tolerance** properties

Property	*Value*
Default Value:	0.0

Data Type:	double
Attribute Description:	Private
Auto. assign value:	FALSE
Comments:	Upper bound on cell delay variance that two consecutive Quality of Service (QoS) class B cells may experience.

Attribute **class_C_CDV_tolerance** properties

Property	*Value*
Default Value:	0.0
Data Type:	double
Attribute Description:	Private
Auto. assign value:	FALSE
Comments:	Upper bound on cell delay variance that two consecutive Quality of Service (QoS) class C cells may experience.

Attribute **class_D_CDV_tolerance** properties

Property	*Value*
Default Value:	0.0
Data Type:	double
Attribute Description:	Private
Auto. assign value:	FALSE
Comments:	Upper bound on cell delay variance that two consecutive Quality of Service (QoS) class D cells may experience. .

Header Block

```
#include "/usr/local/mil3_dir/3.0.A_DL5/models/std/atm/ams_interfaces.h"
#include "/usr/local/mil3_dir/3.0.A_DL5/models/std/atm/ams_aal_interfaces.h"
#include "/usr/work/benton/op_code/badd_interface.h"

/* These are transition conditions. */
#define SIGNAL    ((op_intrpt_type () == OPC_INTRPT_REMOTE) && \
          (op_intrpt_code () == AMSC_INTERFACE_SIGNAL))

#define CALL_REQ_SIGNAL    ((op_intrpt_type () == OPC_INTRPT_REMOTE) && \
                (op_intrpt_code () == BADD_REQUEST_SIGNAL))

#define AAL_CALL_SETUP     ((SIGNAL) &&                              \
                            ((primitive == AMSC_AAL_ESTAB_Req) ||    \
                             (primitive == AMSC_AA_ESTAB_Ind)))

#define SAAL_SIGNAL        ((op_intrpt_type () == OPC_INTRPT_REMOTE) &&\
                           (op_intrpt_code () == AMSC_SAAL_SIGNAL))
```

```
#define AAL_DATA                (op_intrpt_type () == OPC_INTRPT_STRM)

#define EST_CON                 (SIGNAL && (primitive == AMSC_AAL_ESTAB_Con))

#define REL_IND                 (SIGNAL && (primitive == AMSC_AAL_RELEASE_Ind))

#define REL_CON                 (SIGNAL && (primitive == AMSC_AAL_RELEASE_Con))

#define CALL_START              ((op_intrpt_type () == OPC_INTRPT_SELF) &&          \
                                 (op_intrpt_code () == BADD_CALL_SCH_START))

#define CALL_END                ((op_intrpt_type () == OPC_INTRPT_SELF) &&          \
                                 (op_intrpt_code () == AMSC_TGEN_CALL_END))

#define SCHEDULER               ((op_intrpt_type () == OPC_INTRPT_SELF) &&     \
                                 (op_intrpt_code () == BADD_CALL_SCHEDULER))

#define RESCHEDULE              ((op_intrpt_type () == OPC_INTRPT_REMOTE) &&  \
                                 (op_intrpt_code () == BADD_CALL_RESCHEDULE))

#define CHECK_CHANNEL_SIGNAL ((op_intrpt_type () == OPC_INTRPT_REMOTE) &&   \
                                 (op_intrpt_code () == BADD_CHECK_CHANNEL))

#define WAIT_CALL_DURATION  ((op_intrpt_type () == OPC_INTRPT_SELF) &&          \
                    (op_intrpt_code () == AMSC_TGEN_WAIT_CALL_DURATION))

#define  END_SIMULATION     ((op_intrpt_type () == OPC_INTRPT_ENDSIM))

#define NEIGHBOR_NOTIFY     ((op_intrpt_type () == OPC_INTRPT_REMOTE) &&     \
                                 (op_intrpt_code () == AMSC_NEIGHBOR_NOTIFY))

#define NOTIFY_COMPLETE     (ams_neighbor_notify_is_complete (nbr_data_ptr) == OPC_TRUE)

/* These are self interrupt codes. */
#define   BADD_CALL_SCH_START           0
#define AMSC_TGEN_CALL_END              1
#define AMSC_TGEN_DATA_GEN              2
#define AMSC_TGEN_WAIT_CALL_DURATION  3
#define BADD_CALL_SCHEDULER             4
#define BADD_CALL_RESCHEDULE            5
#define BADD_CHECK_CHANNEL              6
#define MAX_CHANNELS                    1024
#define MAXSIZE                         11

/* The scheduler process will output trace information if these conditional are true. */
#define LTRACE_ACTIVE  (debug_mode &&                                       \
                        (op_prg_odb_ltrace_active ("ams") ||                \
                         op_prg_odb_ltrace_active ("ams_traf") ||           \
                         op_prg_odb_ltrace_active ("ams_traf_gen")))

#define LTRACE_CONNECT_ACTIVE     (op_prg_odb_ltrace_active ("connect"))

#define LTRACE_CALL_SCHEDULER_ACTIVE   (op_prg_odb_ltrace_active ("call_scheduler"))

#define LTRACE_CALL_SCH_ACTIVE          (op_prg_odb_ltrace_active ("call_sch"))

#define LTRACE_CALL_DISP_ACTIVE      (op_prg_odb_ltrace_active ("call_disp"))
```

```
                #define LTRACE_CALL_CHANNELS_ACTIVE     (op_prg_odb_ltrace_active ("call_channels"))

                #define LTRACE_CALL_TIMER_ACTIVE        (op_prg_odb_ltrace_active ("call_timer"))
80
                #define LTRACE_CALL_RESCHEDULER_ACTIVE  (op_prg_odb_ltrace_active ("call_rescheduler"))

                /* Function declarations. */
                void badd_call_sch_nbr_intrpt_proc ();
85              void badd_call_sch_spurious_signal_handle ();
                void badd_call_sch_list_print ();
                void badd_call_sch_error ();
```

State Variable Block

```
                int                     \debug_mode;
                int                     \packet_size;
                int                     \dest_addr;
                int                     \AAL_type;
 5              int                     \qos_class;
                int                     \to_aal_stream_index;
                int                     \to_req_stream_index;
                int                     \sch_channel_count;
                int                     \number_calls_pending;
10              int                     \max_calls_pending;
                double                  \avg_rate;
                double                  \channel_delay;
                double                  \trans_delay;
                double                  \time_to_next_scheduler;
15              double                  \end_sim_time;
                char                    \pid_string [128];
                Objid                   \my_id;
                Objid                   \aal_module_id;
                Objid                   \req_module_id;
20              List*                   \calls_pending;
                List*                   \calls_scheduled;
                List*                   \static_channels;
                FILE*                   \output_file;
                FILE*                   \output_calls;
25              Ici*                    \aal_handle_iciptr;
                Evhandle                \next_packet_arrival;
                Evhandle                \call_end_intrpt;
                Distribution*           \int_arrival_distptr;
                Distribution*           \call_wait_distptr;
30              Distribution*           \call_duration_distptr;
                AmsT_Traf_Contract*     \traf_con_ptr;
                AmsT_Neighbor_Data*     \nbr_data_ptr;
                Badd_Sch_Mod_Data       \Scheduler_Module;
                Badd_Channel_Desc*      \channel_ptr;
35
```

Temporary Variable Block

```
                int             primitive;
                int             cd_packet_size;
                int             clark_dest_addr;
                int             calls_list_size;
 5              int             index = 0;
```

```
int                     channel_index;
int                     sch_channel_index;
int                     check_chan_index;
int                     call_bit_rate;
int                     call_compl_channel;
int                     call_release_channel;
int                     total_channels = 0;
int                     channel_count = 0;
int                     value;
int                     least_calls;
int                     least_calls_index;
int                     temp_intrpt_type;
int                     temp_intrpt_code;
int*                    int_ptr;
int*                    channel_size_ptr;
double                  peak_cell_rate;
double                  cdv_tolerance;
double                  int_arr_time;
double                  call_wait_time;
double                  call_duration;
double                  event_time;
double                  channel_compl_time;
double                  temp_double;
double          temp_PCR;
double                  time_enqueued;
double                  time_dequeued;
double                  time_in_queue;
extern int              total_calls_requested;
extern int              total_calls_generated;
extern int              total_calls_received;
extern int              total_calls_active;
extern int              max_calls_active;
char                    string[11];
Ici*                    if_iciptr;
Ici*                    req_iciptr;
Ici*                    call_iciptr;
Ici*                    signal_iciptr;
Ici*                    setup_iciptr;
Ici*                    upper_handle_iciptr;
Ici*                    check_channel_iciptr;
Ici*                    channel_iciptr;
Ici*                    sch_channel_iciptr;
Prohandle               call_gen_prohandle;
Prohandle*              call_gen_proptr;
Evhandle                this_event;
Evhandle                next_event;
Evhandle                scheduler_event;
List*                   channel_listptr;
List*                   sch_channel_listptr;
FILE*                   input_file;
Compcode                channel_scheduled;
Compcode                sch_requested;
AmsT_Traf_Contract*     tmp_traf_con_ptr;
```

Function Block

```
void
badd_call_sch_nbr_intrpt_proc (ndata_ptr, ndesc_ptr, state_ptr)
    AmsT_Neighbor_Data* ndata_ptr;
    AmsT_Neighbor_Desc* ndesc_ptr;
    void*               state_ptr;
    {
    AmsT_Neighbor_Verify_Desc   vdesc;
    int                         to_sw_stream_index;

    /** This procedure handles a neighbor notification event in an AMS          **/
    /** traf gen specific manner. It determines the neighbor's object           **/
    /** ID and type, and verifies that there are a correct number of            **/
    /** interconnections.                                                       **/
    FIN (badd_call_sch_nbr_intrpt_proc (ndata_ptr, ndesc_ptr, state_ptr));

    /* Switch based on the AMS type.                                            */
    switch (ndesc_ptr->module_amstype)
        {
        case AMSC_MTYPE_AAL:
            {
            /* Build the verify desc data structure.                            */
            vdesc.mod_id            = my_id;
            vdesc.nbr_id            = ndesc_ptr->module_objid;
            vdesc.nbr_id_ptr        = &aal_module_id;
            vdesc.mod_name          = "BADD Call Scheduler";
            vdesc.nbr_name          = "AAL";
            vdesc.nbr_type          = AMSC_MTYPE_AAL;
            vdesc.from_nbr_strm_cnt         = 1;
            vdesc.from_nbr_strm_index_ptr   = OPC_NIL;
            vdesc.to_nbr_strm_cnt           = 1;
            vdesc.to_nbr_strm_index_ptr     = &to_aal_stream_index;

            /* Verify that the neighbor has the correct                         */
            /* characteristics.                                                 */
            ams_neighbor_verify (&vdesc);

            break;
            }

        case BADD_MTYPE_SCHEDULER_CLIENT:
            {
            /* Build the verify desc data structure.                            */
            vdesc.mod_id            = my_id;
            vdesc.nbr_id            = ndesc_ptr->module_objid;
            vdesc.nbr_id_ptr        = &req_module_id;
            vdesc.mod_name          = "BADD Call Scheduler";
            vdesc.nbr_name          = "call_req";
            vdesc.nbr_type          = BADD_MTYPE_SCHEDULER_CLIENT;
            vdesc.from_nbr_strm_cnt         = 1;
            vdesc.from_nbr_strm_index_ptr   = OPC_NIL;
            vdesc.to_nbr_strm_cnt           = 1;
            vdesc.to_nbr_strm_index_ptr     = &to_req_stream_index;

            /* Verify that the neighbor has the correct                         */
```

```
                /* characteristics.                                          */
                /* ams_neighbor_verify (&vdesc);          */

                break;
                }

            default:
                {
                /* This is an unexpected neighbor notification.              */
                /* Issue error message.                                      */
                op_sim_end ("Process received a neighbor notification from a module",
                    "of an unexpected type.  Simulation terminated.", OPC_NIL, OPC_NIL);

                break;
                }
            }

        FOUT;
        }

void
badd_call_sch_spurious_signal_handle ()
    {
    Ici*            if_iciptr;
    Ici*            release_if_iciptr;
    int             primitive;
    Ici*            ll_handle_iciptr;
    Packet*         pkptr;

    /* This procedure handles spurious interrupts.                           */
    /* Only three types of spurious interrupts can be accepted. All          */
    /* others must result in an op_sim_end (). These three                   */
    /* interrupts are:                                                       */
    /*   1. An AAL ESTAB Ind.                                                */
    /*   2. An AAL RELEASE Con.                                              */
    /*   3. A packet arrival.                                                */
    FIN (badd_call_sch_spurious_signal_handle ());

    if (AAL_CALL_SETUP)
        {
        /* This is a signal from the AAL.                                    */

        /* Obtain the ICI pointer.                                           */
        if_iciptr = op_intrpt_ici ();

        /* Obtain the primitive from the ICI.                                */
        op_ici_attr_get (if_iciptr, "primitive", &primitive);

        /* Obtain the 'lower layer handle' from the ICI.                     */
        op_ici_attr_get (if_iciptr, "lower layer handle", &ll_handle_iciptr);

        /* Switch on the 'primitive'.          */
        switch (primitive)
            {
            case AMSC_AAL_ESTAB_Ind:
                {
                /* This is an 'establish indication' from the AAL.           */

                if (LTRACE_ACTIVE)
```

```
                    {
115                 op_prg_odb_print_major (pid_string, "Received spurious AAL ESTABLISH Request signal.",
                        "Call not accepted.", "Sending AAL RELEASE Request signal.", OPC_NIL);
                    }

                /* Set the 'upper layer handle' in the interface                */
120             /* ICI for the 'establish indication' signal, since             */
                /* the lower layer expects it to be filled in. The              */
                /* ICI is not destroyed, since this is a forced                 */
                /* interrupt and the lower layer process expects                */
                /* to obtain the handle from the ICI when the                   */
125             /* control returns.                                             */
                op_ici_attr_set (if_iciptr, "upper layer handle", OPC_NIL);

                /* Simply send an AAL_RELEASE_Req to request that               */
                /* the connection be released.                                  */
130             release_if_iciptr = op_ici_create (AMSC_INTERFACE_ICI);
                op_ici_attr_set (release_if_iciptr, "primitive", AMSC_AAL_RELEASE_Req);
                op_ici_attr_set (release_if_iciptr, "lower layer handle", ll_handle_iciptr);
              op_ici_install (release_if_iciptr);

135             /* Send a remote interrupt which will carry the ICI to the AAL module. */
                op_intrpt_schedule_remote (op_sim_time (), AMSC_INTERFACE_SIGNAL, aal_module_id);

                break;
                }
140
            case AMSC_AAL_RELEASE_Con:
                {
                /* This is a 'release confirm' from the AAL.                    */

145             if (LTRACE_ACTIVE)
                    {
                    op_prg_odb_print_major (pid_string, "Received spurious AAL RELEASE Confirm signal.",
                        "This in response to AAL RELEASE Request terminating spurious connection.", O
                    }
150
                /* This is a response to our earlier 'release                   */
                /* request' which was a response to the                         */
                /* spurious 'estab indication'.                                 */
                /* Do nothing other than destroy the ICI.                       */
155             op_ici_destroy (if_iciptr);

                break;
                }

160         default:
                {
                /* This is some other completely unexpected                     */
                /* signal. Issue error message and terminate                    */
                /* simulation.                                                  */
165             op_sim_end ("In badd_call_scheduler, Received unexpected signal.", "", "", "");
                }
            }
        }
    else if (op_intrpt_type () == OPC_INTRPT_STRM)
170     {
        /* This is a spurious packet arrival. The ams_traf_gen                  */
        /* just destroys this packet.                                           */
```

```
            if (LTRACE_ACTIVE)
                {
175             op_prg_odb_print_major (pid_string, "Received spurious DATA packet.",
                    "Destroying the packet.", OPC_NIL);
                }

            /* Get the packet fromt the stream and destroy it.                    */
180         pkptr = op_pk_get (op_intrpt_strm ());
            op_pk_destroy (pkptr);
            }
        else
            {
185         /* This is some other completely unexpected                          */
            /* interrupt. Issue error message and terminate                      */
            /* simulation.                                                       */
            op_sim_end ("Received unexpected interrupt.", "", "", "");
            }
190
        FOUT;
        }

    void
195 badd_call_sch_list_print (calls_list)
      List*     calls_list;
      {
      int   list_size;
      int   index = 0;
200   int   list_packet_size;
      int   list_dest_addr;
      int   list_qos_class;
      int   list_AAL_type;
      double list_bit_rate;
205   double list_int_arr_time;
      double list_call_wait_time;
      double list_call_duration;
      double list_peak_cell_rate;
      Ici*  req_iciptr;
210
      /* This procedure will print all the calls_descs in the calls_pending list */
      FIN (badd_call_sch_list_print(calls_list));

      list_size = op_prg_list_size(calls_list);
215
      if (list_size == 0)
      {
        printf("      Attempting to print empty call_list.\n");
      }
220
      for (index = 0; index < list_size; index++)
      {
        req_iciptr = (Ici*) op_prg_list_access (calls_list, index);

225     if (req_iciptr == OPC_NIL)
          badd_call_sch_error("Unable to get req_iciptr from calls_pending list.", OPC_NIL, OPC_NIL);

        if ((op_ici_attr_get (req_iciptr, "interarrival time", &list_int_arr_time) == OPC_COMPCODE_FAILURE) ||
          (op_ici_attr_get (req_iciptr, "packet size", &list_packet_size) == OPC_COMPCODE_FAILURE)      ||
230       (op_ici_attr_get (req_iciptr, "call wait time", &list_call_wait_time) == OPC_COMPCODE_FAILURE) ||
          (op_ici_attr_get (req_iciptr, "call duration", &list_call_duration) == OPC_COMPCODE_FAILURE)  ||
```

```
        (op_ici_attr_get (req_iciptr, "dest addr", &list_dest_addr) == OPC_COMPCODE_FAILURE)      ||
        (op_ici_attr_get (req_iciptr, "QoS class", &list_qos_class) == OPC_COMPCODE_FAILURE)      ||
        (op_ici_attr_get (req_iciptr, "AAL type", &list_AAL_type) == OPC_COMPCODE_FAILURE)        ||
235     (op_ici_attr_get (req_iciptr, "peak cell rate", &list_peak_cell_rate) == OPC_COMPCODE_FAILURE))
        badd_call_sch_error("Unable to get values from call_req iciptr", OPC_NIL, OPC_NIL);

    printf("In badd_call_scheduler.printing calls_pending list: int_arr_time = %f.\n",
        list_int_arr_time);
240 printf("In badd_call_scheduler.printing calls_pending list: packet_size = %d.\n",
        list_packet_size);
    printf("In badd_call_scheduler.printing calls_pending list: call_wait_time = %f.\n",
        list_call_wait_time);
    printf("In badd_call_scheduler.printing calls_pending list: call_duration = %f.\n",
245     list_call_duration);
    printf("In badd_call_scheduler.printing calls_pending list: dest_addr = %d.\n",
        list_dest_addr);
    printf("In badd_call_scheduler.printing calls_pending list: qos_class = %d.\n",
        list_qos_class);
250 printf("In badd_call_scheduler.printing calls_pending list: AAL_type = %d.\n",
        list_AAL_type);
    printf("In badd_call_scheduler.printing calls_pending list: peak_cell_rate = %f.\n",
        list_peak_cell_rate);
    printf("In badd_call_scheduler.printing calls_pending list: bit_rate = %f.\n",
255     list_peak_cell_rate * 424);
  } /* End for loop */

  FOUT;
  }
260
void
badd_call_sch_error (msg0, msg1, msg2)
    char*       msg0;
    char*       msg1;
265 char*       msg2;
    {
    /** Print an error message and exit the simulation. **/
    FIN (badd_call_sch_error (msg0, msg1, msg2));

270 op_sim_end ("Error in badd_call_scheduler process:",
        msg0, msg1, msg2);

    FOUT;
    }
275
```

Diagnostic Block

```
/* Display connection information, if requested. */
if (LTRACE_CONNECT_ACTIVE)
    {
5   /* Print information. */
    ams_neighbor_data_print (nbr_data_ptr, ams_neighbor_desc_print_noop);
    }
```

...
...

forced state **INIT**

attribute	*value*	*type*	*default value*
name	INIT	string	st
enter execs	(See below.)	textlist	
exit execs	(empty)	textlist	(empty)
status	forced	toggle	unforced

enter execs **INIT**

```
   /* Determine the initial values of the state variables and set up          */
   /* the initial state of this instantiation of the ams_traf_gen             */
   /* process.                                                                */

 5 /* Obtain the object ID of this process' parent module.                    */
   my_id = op_id_self ();

   /* Determine whether or not the simulation is in debug mode.               */
   debug_mode = op_sim_debug ();
10
   /* Initialize the AAL module object ID to NULL value.                      */
   aal_module_id = OPC_OBJID_NULL;
   req_module_id = OPC_OBJID_NULL;

15 /* Create a list for storing the calls as they arrive */
   calls_pending = op_prg_list_create();

   /* Read the static channels input file and create lists accordingly. */
   static_channels = op_prg_list_create();
20 calls_scheduled = op_prg_list_create();

   /* Open the channel definition file. */
   if ((input_file = fopen("/usr/work/benton/input_channels", "r"))
                                        == NULL)
25 {
     badd_call_sch_error("Problem opening static channel definition file.",
                                 OPC_NIL, OPC_NIL);
   }
   if (fgets ( string, MAXSIZE, input_file ) != NULL)
30 {
     if (LTRACE_CALL_CHANNELS_ACTIVE)
       printf("In badd_call_sch.init; string is %s.\n", string);
     if (get_digit(string, &value) == OPC_COMPCODE_FAILURE)
     {
35     badd_call_sch_error ("Invalid number of static channels.",
                 OPC_NIL, OPC_NIL);
     }
     total_channels = value;
     if (LTRACE_CALL_CHANNELS_ACTIVE)
40   {
       printf("In badd_call_sch.load_static_channels function after reading channel");
       printf(" count;total_channels = %d.\n", total_channels);
     }
   }
45 if (total_channels > MAX_CHANNELS)
   {
```

```
    badd_call_sch_error("Requested virtual channels exceeds maximum allowed.",
              OPC_NIL, OPC_NIL);
  }

  while ((fgets ( string, MAXSIZE, input_file ) != NULL) &&
      (channel_count < total_channels))
  {
    if (LTRACE_CALL_CHANNELS_ACTIVE)
    {
      printf("In badd_call_sch.init, reading each channel size;");
      printf(" channel_count = %d.\n", channel_count);
      printf("In badd_call_sch.init; string value is %s.\n", string);
    }
    if (get_digit(string, &value) == OPC_COMPCODE_FAILURE)
    {
      badd_call_sch_error ("Invalid number for static channels.",
                OPC_NIL, OPC_NIL);
    }
    if (value > 8000000)
    {
      badd_call_sch_error (
    }
    channel_ptr = (Badd_Channel_Desc*) op_prg_mem_alloc(sizeof(Badd_Channel_Desc));
    if (channel_ptr == OPC_NIL)
    {
      badd_call_sch_error("Unable to allocate memory for channel definition.",
                OPC_NIL, OPC_NIL);
    }
    channel_ptr->ch_capacity = value;
    channel_ptr->ch_calls_sch = -1;
    channel_ptr->ch_calls_compl = 0;
    channel_ptr->ch_compl_time = 0.0;

    op_prg_list_insert(static_channels, channel_ptr, OPC_LISTPOS_TAIL);
    if (LTRACE_CALL_CHANNELS_ACTIVE)
    {
      printf("In badd_call_sch.init, channel value saved = %d.\n", channel_ptr->ch_capacity);
    }

    /* Create a list to store the calls for this channel */
    channel_listptr = op_prg_list_create();
    if (channel_listptr == OPC_NIL)
    {
      badd_call_sch_error("Unable to allocate memory for channel list.",
                OPC_NIL, OPC_NIL);
    }
    op_prg_list_insert(calls_scheduled, channel_listptr, OPC_LISTPOS_TAIL);
    channel_count = channel_count + 1;
  }
  if (LTRACE_CALL_CHANNELS_ACTIVE)
  {
    channel_count = op_prg_list_size(static_channels);
    printf("In badd_call_sch.init, elements in static channels ");
    printf("list = %d.\n", channel_count);
    channel_count = op_prg_list_size(calls_scheduled);
    printf("In badd_call_sch.init, elements in sch_channels ");
    printf("list = %d.\n", channel_count);
  }
```

```
    fclose(input_file);

    /* Open file for storing call timer information. */
110 if ((output_file = fopen("/usr/work/benton/output_call_times_fifo", "w"))
                                        == NULL)
    {
      badd_call_sch_error("Problem opening output_call_times file.",
                                    OPC_NIL, OPC_NIL);
115 }

    /* Write a file format statement as the first line in the file. */
    fputs("File Format: Channel Enqueued_Time Dequeued_Time Time_In_Queue PCR\n",
        output_file);
120
    /* Open file for storing calls completed information. */
    if ((output_calls = fopen("/usr/work/benton/output_call_completed_fifo", "w"))
                                        == NULL)
    {
125   badd_call_sch_error("Problem opening output_call_completed file.",
                                    OPC_NIL, OPC_NIL);
    }

    /* Write a file format statement as the first line in the file. */
130 fputs("File Format: Channel Time_completed PCR.\n", output_calls);

    sch_channel_count = op_prg_list_size(static_channels);

    /* Generate PID display string.                                    */
135 sprintf (pid_string, "badd_call_scheduler PID (%d)", op_pro_id (op_pro_self ()));

    /* Obtain and send out neighbor information.                       */
    nbr_data_ptr = ams_neighbor_data_build ();
    ams_neighbor_notify (nbr_data_ptr, AMSC_MTYPE_AAL_CLIENT);
140
    if (LTRACE_CALL_SCHEDULER_ACTIVE)
    {
      printf("In badd_call_scheduler.init: print neighbor info.\n");
      ams_neighbor_data_print (nbr_data_ptr, ams_neighbor_desc_print_noop);
145 }

    /* This Scheduler_Module is attached to any process spawned by */
    /* this badd_call_requestor process.                           */
    /*Scheduler_Module = (Badd_Sch_Mod_Data*) op_prg_mem_alloc(sizeof(Badd_Sch_Mod_Data)); */
150 op_pro_modmem_install(&Scheduler_Module);

    /* Initialize the model parameter attributes. */
    op_ima_obj_attr_get (my_id, "scheduler delay", &time_to_next_scheduler);
    op_ima_obj_attr_get (my_id, "transmission delay", &trans_delay);
155 op_ima_obj_attr_get (my_id, "channel delay", &channel_delay);
    op_ima_obj_attr_get (my_id, "calls pending", &max_calls_pending);

    /* Initialize the end_sim_time varaible */
    op_ima_sim_attr_get(OPC_IMA_DOUBLE, "duration", &end_sim_time);
160 /* printf("end_sim_time = %f.\n", end_sim_time); */

    if (LTRACE_CALL_SCHEDULER_ACTIVE)
    {

165   printf("In badd_call_scheduler.init state: Leaving init state. \n");
```

...
...

```
}
```

transition **INIT -> config**

attribute	*value*	*type*	*default value*
name	tr_1	string	tr
condition		string	
executive		string	
color	RGB333	color	RGB333
drawing style	line	toggle	spline

unforced state **config**

attribute	*value*	*type*	*default value*
name	config	string	st
enter execs	(empty)	textlist	(empty)
exit execs	(See below.)	textlist	
status	unforced	toggle	unforced

exit execs **config**

```
    /* Ams_traf_gen expects either a neighbor notification interrupt,          */
    /* or a spurious signal.                                                   */
    if (NEIGHBOR_NOTIFY)
        {
5       /* This is a 'neighbor notify' signal.                                 */
        if (LTRACE_ACTIVE)
            {
            op_prg_odb_print_major (pid_string, "Received neighbor notification.", OPC_NIL);
            }
10
        /* Handle the neighbor notification.                                   */
        ams_neighbor_interrupt_handle (nbr_data_ptr, badd_call_sch_nbr_intrpt_proc, OPC_NIL);
        }
    else
15      {
        /* This is a spurious interrupt. Handle appropriately.                 */
        badd_call_sch_spurious_signal_handle ();
        }
```

transition **config -> config**

attribute	*value*	*type*	*default value*
name	tr_0	string	tr
condition	default	string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

...

...

transition **config -> shared data**

attribute	*value*	*type*	*default value*
name	tr_118	string	tr
condition	NOTIFY_COMPLETE	string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

unforced state **dispatch**

attribute	*value*	*type*	*default value*
name	dispatch	string	st
enter execs	(See below.)	textlist	
exit execs	(See below.)	textlist	
status	unforced	toggle	unforced

enter execs **dispatch**

```
   /* Compare the number of calls in the calls_pending list. */
   /* If the number exceeds the max_calls_pending, schedule  */
   /* a call to execute the scheduler.                       */
   number_calls_pending = op_prg_list_size(calls_pending);
5  if (number_calls_pending > max_calls_pending)
     op_intrpt_schedule_self (op_sim_time (), BADD_CALL_SCHEDULER);

   /* Schedule the next scheduling event */
   if (number_calls_pending > 0)
10   op_intrpt_schedule_self (op_sim_time () + time_to_next_scheduler, BADD_CALL_SCHEDULER);
```

exit execs **dispatch**

```
   temp_intrpt_type = op_intrpt_type();
   temp_intrpt_code = op_intrpt_code();

   if (LTRACE_CALL_DISP_ACTIVE)
5  {
     printf("In badd_call_scheduler.dispatch: received SIGNAL.\n");
     printf("In badd_call_scheduler.dispatch: intrpt_type = %d.\n", temp_intrpt_type);
     printf("In badd_call_scheduler.dispatch: intrpt_code = %d.\n", temp_intrpt_code);
   } /* End if (LTRACE_CALL_DISP_ACTIVE) */
10
   if (SIGNAL)
   {
     signal_iciptr = op_intrpt_ici();
     if (signal_iciptr == OPC_NIL)
15     badd_call_sch_error("Unable to get signal_iciptr.", OPC_NIL, OPC_NIL);

     if (LTRACE_CALL_DISP_ACTIVE)
     {
       op_ici_print(signal_iciptr);
```

...
...

```
20    } /* End if (LTRACE_CALL_DISP_ACTIVE) */

      op_ici_attr_get(signal_iciptr, "primitive", &primitive);
    }
```

transition **dispatch -> call request**			
attribute	*value*	*type*	*default value*
name	tr_111	string	tr
condition	CALL_REQ_SIGNAL	string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

transition **dispatch -> send data**			
attribute	*value*	*type*	*default value*
name	tr_131	string	tr
condition	EST_CON	string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

transition **dispatch -> call complete**			
attribute	*value*	*type*	*default value*
name	tr_136	string	tr
condition	REL_CON	string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

transition **dispatch -> call released**			
attribute	*value*	*type*	*default value*
name	tr_139	string	tr
condition	REL_IND	string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

transition **dispatch -> call schedule**			
attribute	*value*	*type*	*default value*
name	tr_142	string	tr
condition	SCHEDULER	string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

...
...

transition **dispatch -> End Sim**			
attribute	*value*	*type*	*default value*
name	tr_145	string	tr
condition	END_SIMULATION	string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

transition **dispatch -> reschedule**			
attribute	*value*	*type*	*default value*
name	tr_148	string	tr
condition	RESCHEDULE	string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

transition **dispatch -> check channel**			
attribute	*value*	*type*	*default value*
name	tr_151	string	tr
condition	CHECK_CHANNEL_SIGNAL	string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

transition **dispatch -> start call**			
attribute	*value*	*type*	*default value*
name	tr_154	string	tr
condition	CALL_START	string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

transition **dispatch -> error**			
attribute	*value*	*type*	*default value*
name	tr_157	string	tr
condition	default	string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

forced state **start call**			
attribute	*value*	*type*	*default value*
name	start call	string	st

enter execs	(See below.)	textlist	
exit execs	(empty)	textlist	(empty)
status	forced	toggle	unforced

enter execs **start call**

```
sch_channel_iciptr = op_intrpt_ici();
if ((op_ici_attr_get (sch_channel_iciptr, "channel number", &sch_channel_index) == OPC_COMPCODE_FAILURE))
  badd_call_sch_error("In start_call: unable to get values from sch_channel iciptr",
            OPC_NIL, OPC_NIL);

/* Destroy the ici to recover space, Garbage collection. */
op_ici_destroy(sch_channel_iciptr);

if (LTRACE_CALL_SCH_ACTIVE)
  printf("In call scheduler.start_call: sch_channel index = %d.\n", sch_channel_index);
sch_channel_listptr = (List*) op_prg_list_access(calls_scheduled, sch_channel_index);
call_iciptr = (Ici*) op_prg_list_remove(sch_channel_listptr, OPC_LISTPOS_HEAD);

if (call_iciptr == OPC_NIL)
  badd_call_sch_error("In start call, unable to get call_iciptr from calls_list.", OPC_NIL, OPC_NIL);

if ((op_ici_attr_get (call_iciptr, "interarrival time", &int_arr_time) == OPC_COMPCODE_FAILURE) ||
  (op_ici_attr_get (call_iciptr, "packet size", &packet_size) == OPC_COMPCODE_FAILURE)     ||
  (op_ici_attr_get (call_iciptr, "call wait time", &call_wait_time) == OPC_COMPCODE_FAILURE) ||
  (op_ici_attr_get (call_iciptr, "call duration", &call_duration) == OPC_COMPCODE_FAILURE)  ||
  (op_ici_attr_get (call_iciptr, "dest addr", &dest_addr) == OPC_COMPCODE_FAILURE)        ||
  (op_ici_attr_get (call_iciptr, "QoS class", &qos_class) == OPC_COMPCODE_FAILURE)        ||
  (op_ici_attr_get (call_iciptr, "AAL type", &AAL_type) == OPC_COMPCODE_FAILURE)          ||
  (op_ici_attr_get (call_iciptr, "peak cell rate", &peak_cell_rate) == OPC_COMPCODE_FAILURE) ||
  (op_ici_attr_get (call_iciptr, "time queued", &time_enqueued) == OPC_COMPCODE_FAILURE))
  badd_call_sch_error("Unable to get values from call_req iciptr", OPC_NIL, OPC_NIL);

if (LTRACE_CALL_SCHEDULER_ACTIVE)
{
  printf("In badd_call_scheduler.start_call: int_arr_time = %f.\n", int_arr_time);
  printf("In badd_call_scheduler.start_call: packet_size = %d.\n", packet_size);
  printf("In badd_call_scheduler.start_call: call_wait_time = %f.\n", call_wait_time);
  printf("In badd_call_scheduler.start_call: call_duration = %f.\n", call_duration);
  printf("In badd_call_scheduler.start_call: dest_addr = %d.\n", dest_addr);
  printf("In badd_call_scheduler.start_call: qos_class = %d.\n", qos_class);
  printf("In badd_call_scheduler.start_call: AAL_type = %d.\n", AAL_type);
  printf("In badd_call_scheduler.start_call: peak_cell_rate = %f.\n", peak_cell_rate);
} /* End if (LTRACE_CALL_SCHEDULER_ACTIVE) */

if (op_ici_attr_set (call_iciptr, "channel assigned", sch_channel_index) == OPC_COMPCODE_FAILURE)
  badd_call_sch_error("Unable to set sch_channel_index in call_iciptr", OPC_NIL, OPC_NIL);

time_dequeued = op_sim_time();
time_in_queue = time_dequeued - time_enqueued;

/* Write queue times to the call timer output file. */
fprintf(output_file, "%d %f %f %f %f\n", sch_channel_index, time_enqueued, time_dequeued,
    time_in_queue, peak_cell_rate);

/* Update program counter */
total_calls_generated = total_calls_generated + 1;
total_calls_active = total_calls_active + 1;
```

```
...
...
    if (total_calls_active > max_calls_active)
      max_calls_active = total_calls_active;
55
    if (LTRACE_CALL_SCHEDULER_ACTIVE)
    {
      printf("In badd_call_scheduler, start; starting call to dest %d.\n",
           dest_addr);
60  } /* if (LTRACE_CALL_SCHEDULER_ACTIVE) */

    /* Spawn a child process to generate the call data */
    call_gen_prohandle = op_pro_create("clark_badd_call_generator", OPC_NIL);
    if (op_pro_valid(call_gen_prohandle) == OPC_FALSE)
65  {
     badd_call_sch_error("Unable to create call generator process", OPC_NIL, OPC_NIL);
    }

    if (LTRACE_CALL_SCHEDULER_ACTIVE)
70  {
      printf("In badd_call_scheduler, start; invoking call generator.\n");
      printf("In badd_call_scheduler.start: call_iciptr = %x.\n", call_iciptr);
      printf("In badd_call_scheduler.start_call; md_aal_module_id = %d.\n", Scheduler_Module.md_aal_module_id);

75  } /* if (LTRACE_CALL_SCHEDULER_ACTIVE) */

    if (LTRACE_CALL_TIMER_ACTIVE)
      printf("In badd_call_sch.start call: current time = %f.\n", op_sim_time());

80  /* When invoking the process, pass in the call desc */
    if (op_pro_invoke(call_gen_prohandle, call_iciptr) == OPC_COMPCODE_FAILURE)
    {
     badd_call_sch_error("Unable to invoke call generator process", OPC_NIL, OPC_NIL);
    }
85
```

transition **start call -> dispatch**			
attribute	*value*	*type*	*default value*
name	tr_155	string	tr
condition		string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

forced state **send data**			
attribute	*value*	*type*	*default value*
name	send data	string	st
enter execs	(See below.)	textlist	
exit execs	(empty)	textlist	(empty)
status	forced	toggle	unforced

...
...

enter execs **send data**

```
signal_iciptr = op_intrpt_ici();

if (signal_iciptr == OPC_NIL)
  badd_call_sch_error("Unable to get signal_iciptr.", OPC_NIL, OPC_NIL);

if (LTRACE_CALL_SCHEDULER_ACTIVE)
{
  printf("In badd_call_scheduler.send data: restarting call_gen.\n");
  op_ici_print(signal_iciptr);

} /* End if (LTRACE_CALL_SCHEDULER_ACTIVE) */

if (op_ici_attr_get(signal_iciptr, "upper layer handle", &upper_handle_iciptr) == OPC_COMPCODE_FAILURE)
  badd_call_sch_error("Unable to get upper layer handle.", OPC_NIL, OPC_NIL);

if (op_ici_attr_get(upper_handle_iciptr, "call_gen_prohandle", &call_gen_proptr) == OPC_COMPCODE_FAILURE)
  badd_call_sch_error("Unable to get call_gen prohandle.", OPC_NIL, OPC_NIL);

if (LTRACE_CALL_TIMER_ACTIVE)
  printf("In badd_call_sch.send data: current time = %f.\n", op_sim_time());

if (op_pro_invoke(*call_gen_proptr, signal_iciptr) == OPC_COMPCODE_FAILURE)
{
  badd_call_sch_error("Unable to restart call_gen process", OPC_NIL, OPC_NIL);
}
```

transition **send data -> dispatch**

attribute	*value*	*type*	*default value*
name	tr_132	string	tr
condition		string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

forced state **call complete**

attribute	*value*	*type*	*default value*
name	call complete	string	st
enter execs	(See below.)	textlist	
exit execs	(empty)	textlist	(empty)
status	forced	toggle	unforced

enter execs **call complete**

```
if (signal_iciptr == OPC_NIL)
  badd_call_sch_error("Unable to get signal_iciptr.", OPC_NIL, OPC_NIL);

if (LTRACE_CALL_SCHEDULER_ACTIVE)
{
  printf("In badd_call_scheduler.call_complete: restarting call_gen.\n");
```

```
    op_ici_print(signal_iciptr);

  } /* End if(LTRACE_CALL_SCHEDULER_ACTIVE) */
10
  total_calls_received = total_calls_received + 1;
  total_calls_active = total_calls_active - 1;

  if (op_ici_attr_get(signal_iciptr, "upper layer handle", &upper_handle_iciptr) == OPC_COMPCODE_FAILURE)
15  badd_call_sch_error("Unable to get upper layer handle.", OPC_NIL, OPC_NIL);

  if (op_ici_attr_get(signal_iciptr, "traffic contract", &tmp_traf_con_ptr) == OPC_COMPCODE_FAILURE)
    badd_call_sch_error("Unable to get traffic contract.", OPC_NIL, OPC_NIL);

20 if (op_ici_attr_get(signal_iciptr, "badd_call_gen_channel_id", &call_compl_channel) == OPC_COMPCODE_FAILURE)
    badd_call_sch_error("Unable to get call_gen prohandle.", OPC_NIL, OPC_NIL);

  /* Write completion time and PCR to the calls completed output file. */
  temp_PCR = tmp_traf_con_ptr->calling_ctd.src_traf_desc.pcr;
25 fprintf(output_calls, "%d %f %f\n", call_compl_channel, op_sim_time(), temp_PCR);

  channel_ptr = (Badd_Channel_Desc*) op_prg_list_access(static_channels, call_compl_channel);
  channel_ptr->ch_calls_sch = channel_ptr->ch_calls_sch - 1;
  channel_ptr->ch_calls_compl = channel_ptr->ch_calls_compl + 1;
30
  if (op_ici_attr_get(upper_handle_iciptr, "call_gen_prohandle", &call_gen_proptr) == OPC_COMPCODE_FAILURE)
    badd_call_sch_error("Unable to get call_gen prohandle.", OPC_NIL, OPC_NIL);

  if (LTRACE_CALL_TIMER_ACTIVE)
35  printf("In badd_call_sch.call complete: current time = %f.\n", op_sim_time());

  if (op_pro_invoke(*call_gen_proptr, signal_iciptr) == OPC_COMPCODE_FAILURE)
  {
   badd_call_sch_error("Unable to restart call_gen process", OPC_NIL, OPC_NIL);
40 }
```

transition **call complete -> dispatch**			
attribute	*value*	*type*	*default value*
name	tr_137	string	tr
condition		string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

forced state **call released**			
attribute	*value*	*type*	*default value*
name	call released	string	st
enter execs	(See below.)	textlist	
exit execs	(empty)	textlist	(empty)
status	forced	toggle	unforced

...
...

enter execs **call released**

```
if (signal_iciptr == OPC_NIL)
  badd_call_sch_error("Unable to get signal_iciptr.", OPC_NIL, OPC_NIL);

if (LTRACE_CALL_SCHEDULER_ACTIVE)
{
  printf("In badd_call_scheduler.call_released: restarting call_gen.\n");
  op_ici_print(signal_iciptr);

} /* End if (LTRACE_CALL_SCHEDULER_ACTIVE) */

if (op_ici_attr_get(signal_iciptr, "upper layer handle", &upper_handle_iciptr) == OPC_COMPCODE_FAILURE)
  badd_call_sch_error("Unable to get upper layer handle.", OPC_NIL, OPC_NIL);

if (op_ici_attr_get(signal_iciptr, "badd_call_gen_channel_id", &call_release_channel) == OPC_COMPCODE_FAILURE)
  badd_call_sch_error("Unable to get call_gen call_release_channel.", OPC_NIL, OPC_NIL);

channel_ptr = (Badd_Channel_Desc*) op_prg_list_access(static_channels, call_release_channel);
channel_ptr->ch_calls_sch = channel_ptr->ch_calls_sch - 1;

if (op_ici_attr_get(upper_handle_iciptr, "call_gen_prohandle", &call_gen_proptr) == OPC_COMPCODE_FAILURE)
  badd_call_sch_error("Unable to get call_gen prohandle.", OPC_NIL, OPC_NIL);

total_calls_active = total_calls_active - 1;

if (op_pro_invoke(*call_gen_proptr, signal_iciptr) == OPC_COMPCODE_FAILURE)
{
  badd_call_sch_error("Unable to restart call_gen process", OPC_NIL, OPC_NIL);
}
```

transition **call released -> dispatch**

attribute	*value*	*type*	*default value*
name	tr_140	string	tr
condition		string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

forced state **shared data**

attribute	*value*	*type*	*default value*
name	shared data	string	st
enter execs	(See below.)	textlist	
exit execs	(empty)	textlist	(empty)
status	forced	toggle	unforced

enter execs **shared data**

```
/* This Scheduler_Module is attached to any process spawned by */
/* this badd_call_scheduler process.                           */
op_pro_modmem_install(&Scheduler_Module);
```

```
/* Pass the neighbor information to all children processes.  */
Scheduler_Module.md_neighbor_data_ptr = nbr_data_ptr;
Scheduler_Module.md_aal_module_id = aal_module_id;
Scheduler_Module.md_to_aal_stream_index = to_aal_stream_index;
Scheduler_Module.md_calls_pending_ptr = calls_pending;
Scheduler_Module.md_channel_delay = channel_delay;
```

transition **shared data -> dispatch**

attribute	*value*	*type*	*default value*
name	tr_121	string	tr
condition		string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

unforced state **error**

attribute	*value*	*type*	*default value*
name	error	string	st
enter execs	(See below.)	textlist	
exit execs	(empty)	textlist	(empty)
status	unforced	toggle	unforced

enter execs **error**

```
/* This is a spurious interrupt. Handle appropriately.        */

printf("Bad signal received by badd_call_scheduler.\n");
badd_call_sch_spurious_signal_handle ();
```

unforced state **End Sim**

attribute	*value*	*type*	*default value*
name	End Sim	string	st
enter execs	(See below.)	textlist	
exit execs	(See below.)	textlist	
status	unforced	toggle	unforced

enter execs **End Sim**

```
/* Print Information during testing */
printf("Calls currently active is %d.\n", total_calls_active);
printf("Max calls active in this run was %d.\n", max_calls_active);

calls_list_size = op_prg_list_size(calls_pending);
```

```
    printf("Calls remaining in the calls_pending list at termination = %d.\n", calls_list_size);

    printf("Calls remaining in the calls_scheduled lists at termination.\n");
10
    time_dequeued = end_sim_time;

    for (channel_index = 0; channel_index < sch_channel_count; channel_index++)
    {
15    channel_ptr = (Badd_Channel_Desc*) op_prg_list_access(static_channels, channel_index);
      calls_list_size = channel_ptr->ch_calls_sch;
      call_compl_channel = channel_ptr->ch_calls_compl;
      channel_compl_time = channel_ptr->ch_compl_time;
      printf("      Channel %d: calls compl = %d, calls remaining = %d, completion time = %f.\n",
20        channel_index, call_compl_channel, calls_list_size, channel_compl_time);
      channel_listptr = (List*) op_prg_list_access(calls_scheduled, channel_index);

      for (sch_channel_index = 0; sch_channel_index < calls_list_size; sch_channel_index++)
      {
25      call_iciptr = (Ici*) op_prg_list_access(channel_listptr, sch_channel_index);
        if ((op_ici_attr_get (call_iciptr, "time queued", &time_enqueued) == OPC_COMPCODE_FAILURE))
          badd_call_sch_error("In End Sim: unable to get values from call_req iciptr",
                    OPC_NIL, OPC_NIL);

30      time_in_queue = time_dequeued - time_enqueued;

        /* Write queue times to the call timer output file. */
        fprintf(output_file, "%d %f %f %f\n", channel_index, time_enqueued, 0.0, 0.0);
/*      fprintf(output_file, "%d %f %f %f\n", channel_index, time_enqueued, time_dequeued,
35 /*          time_in_queue); */
      }

      if (LTRACE_CALL_SCH_ACTIVE)
      {
40      badd_call_sch_list_print(channel_listptr);
      }
    }

    /* Close the Output files. */
45  fclose(output_file);
    fclose(output_calls);

    badd_call_sch_error("Ending simulation in badd_call_scheduler.End Sim.\n", OPC_NIL, OPC_NIL);
```

exit execs **End Sim**

forced state **call request**			
attribute	*value*	*type*	*default value*
name	call request	string	st
enter execs	(See below.)	textlist	
exit execs	(empty)	textlist	(empty)

status	forced	toggle	unforced

enter execs **call request**

```
/* A call_request arrived from the call_requestor above. */
/* Must capture the call_request information from the    */
/* ICI and store the call in the call_pending_list      */

req_iciptr = op_intrpt_ici();

if (req_iciptr == OPC_NIL)
  badd_call_sch_error("Unable to get call_req iciptr.", OPC_NIL, OPC_NIL);

if ((op_ici_attr_get (req_iciptr, "interarrival time", &int_arr_time) == OPC_COMPCODE_FAILURE) ||
  (op_ici_attr_get (req_iciptr, "packet size", &packet_size) == OPC_COMPCODE_FAILURE)      ||
  (op_ici_attr_get (req_iciptr, "call wait time", &call_wait_time) == OPC_COMPCODE_FAILURE) ||
  (op_ici_attr_get (req_iciptr, "call duration", &call_duration) == OPC_COMPCODE_FAILURE)  ||
  (op_ici_attr_get (req_iciptr, "dest addr", &dest_addr) == OPC_COMPCODE_FAILURE)          ||
  (op_ici_attr_get (req_iciptr, "QoS class", &qos_class) == OPC_COMPCODE_FAILURE)          ||
  (op_ici_attr_get (req_iciptr, "AAL type", &AAL_type) == OPC_COMPCODE_FAILURE)            ||
  (op_ici_attr_get (req_iciptr, "peak cell rate", &peak_cell_rate) == OPC_COMPCODE_FAILURE))
  badd_call_sch_error("In badd_call_sch.call_req: unable to get values from call_req iciptr", OPC_NIL, O

/* Destroy the ici to recover space, garbage collection. */
op_ici_destroy(req_iciptr);

if (LTRACE_CALL_SCHEDULER_ACTIVE)
{
  printf("In badd_call_scheduler.call_request: int_arr_time = %f.\n", int_arr_time);
  printf("In badd_call_scheduler.call_request: packet_size = %d.\n", packet_size);
  printf("In badd_call_scheduler.call_request: call_wait_time = %f.\n", call_wait_time);
  printf("In badd_call_scheduler.call_request: call_duration = %f.\n", call_duration);
  printf("In badd_call_scheduler.call_request: dest_addr = %d.\n", dest_addr);
  printf("In badd_call_scheduler.call_request: qos_class = %d.\n", qos_class);
  printf("In badd_call_scheduler.call_request: AAL_type = %d.\n", AAL_type);
  printf("In badd_call_scheduler.call_request: peak_cell_rate = %f.\n", peak_cell_rate);
  printf("In badd_call_scheduler.call_request: req_iciptr = %x.\n", req_iciptr);
} /* End if (LTRACE_CALL_SCHEDULER_ACTIVE) */

/* Create and set the fields in the interface ICI.                */
/* -- Using local memory --                                       */
if_iciptr = op_ici_create ("badd_call_req_if_ici");

op_ici_attr_set (if_iciptr, "interarrival time", int_arr_time);
op_ici_attr_set (if_iciptr, "packet size", packet_size);
op_ici_attr_set (if_iciptr, "call wait time", call_wait_time);
op_ici_attr_set (if_iciptr, "call duration", call_duration);
op_ici_attr_set (if_iciptr, "dest addr", dest_addr);
op_ici_attr_set (if_iciptr, "QoS class", qos_class);
op_ici_attr_set (if_iciptr, "AAL type", AAL_type);
op_ici_attr_set (if_iciptr, "peak cell rate", peak_cell_rate);

op_prg_list_insert(calls_pending, if_iciptr, OPC_LISTPOS_TAIL);

total_calls_requested = total_calls_requested + 1;

/* Send an intrpt to start the scheduler if not requested. */
sch_requested = OPC_COMPCODE_FAILURE;
```

```
55
   this_event = op_ev_current();
   next_event = op_ev_next_local(this_event);
   while (op_ev_valid(next_event))
   {
60   if ((op_ev_type(next_event) == OPC_INTRPT_SELF) &&
       (op_ev_code(next_event) == BADD_CALL_SCHEDULER))
     {
       if (LTRACE_CALL_SCH_ACTIVE)
         printf("Found scheduler event, stopping search.\n");
65     sch_requested = OPC_COMPCODE_SUCCESS;
       break;
     }
     else
       next_event = op_ev_next_local(next_event);
70 }

   /*if (sch_requested == OPC_COMPCODE_FAILURE)
   /*{
   /*   if (LTRACE_CALL_SCH_ACTIVE)
75 /*      printf("Sending for scheduler interrupt.\n");
   /*   op_intrpt_schedule_self (op_sim_time (), BADD_CALL_SCHEDULER);
   /*} */
```

transition **call request -> dispatch**			
attribute	*value*	*type*	*default value*
name	tr_113	string	tr
condition		string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

forced state **call schedule**			
attribute	*value*	*type*	*default value*
name	call schedule	string	st
enter execs	(See below.)	textlist	
exit execs	(empty)	textlist	(empty)
status	forced	toggle	unforced

enter execs **call schedule**

```
   /* Disable the intrpts to make this process atomic. */
   op_intrpt_disable(OPC_INTRPT_SELF, BADD_CALL_SCHEDULER, OPC_FALSE);

   /* Remove all the other events of this type from the events list. */
 5 this_event = op_ev_current();
   next_event = op_ev_next_local(this_event);
   while (op_ev_valid(next_event))
   {
     if ((op_ev_type(next_event) == OPC_INTRPT_SELF) &&
10     (op_ev_code(next_event) == BADD_CALL_SCHEDULER))
```

```
    {
      if (LTRACE_CALL_SCH_ACTIVE)
        printf("Found next sch event, deleting event.\n");
      scheduler_event = next_event;
      next_event = op_ev_next_local(scheduler_event);
      op_ev_cancel(scheduler_event);
    }
    else
      next_event = op_ev_next_local(next_event);
}

/* Determine the number of calls that must be scheduled. */
calls_list_size = op_prg_list_size(calls_pending);

if (LTRACE_CALL_SCH_ACTIVE)
  {
   if (calls_list_size == 0)
     printf("In badd_call_scheduler.call scheduler, attempted to schedule empty list.");
   else
     printf("In badd_call_scheduler.call scheduler, calls list size = %d.\n", calls_list_size);
  }

/* Attempt to schedule each call in the calls_pending list. */
for (index = 0; index < calls_list_size; index++)
{

  if (LTRACE_CALL_SCH_ACTIVE)
  {
   printf("In badd_call_scheduler.call scheduler, index = %d.\n", index);
  }

  /* Get a call description from the calls_pending list. */
  call_iciptr = (Ici*) op_prg_list_remove(calls_pending, OPC_LISTPOS_HEAD);

  if (call_iciptr == OPC_NIL)
    badd_call_sch_error("In badd_call_scheduler.call schedule, unable to get call_iciptr from calls_lis

  if ((op_ici_attr_get (call_iciptr, "peak cell rate", &peak_cell_rate) == OPC_COMPCODE_FAILURE))
    badd_call_sch_error("In badd_sch.call schedule: unable to get values from call_req iciptr.", OPC_NIL

  if (LTRACE_CALL_SCH_ACTIVE)
  {
   printf("In badd_call_scheduler.call schedule: peak_cell_rate = %f.\n", peak_cell_rate);
  } /* End if (LTRACE_CALL_SCHEDULER_ACTIVE) */

  call_bit_rate = peak_cell_rate * AMSC_ATM_CELL_SIZE;

  channel_scheduled = OPC_COMPCODE_FAILURE;

  least_calls = 999999;
  least_calls_index = -1;

  /* Find the channels that can support this call and schedule this call on the */
  /* channel with the least number of calls currently scheduled.                */
  for (channel_index = 0; channel_index < sch_channel_count; channel_index++)
  {
    channel_ptr = (Badd_Channel_Desc*) op_prg_list_access(static_channels, channel_index);
    if (LTRACE_CALL_SCH_ACTIVE)
    {
```

```
70          printf("In badd_call_scheduler.call scheduler, ch_capacity = %d.\n", channel_ptr->ch_capacity);
            printf("In badd_call_scheduler.call scheduler, bit rate = %d.\n", call_bit_rate);
        }

        /* Determine if this channel can support the call */
75      if (call_bit_rate <= channel_ptr->ch_capacity)
        {
          if (least_calls_index < 0)
            least_calls_index = channel_index;

80        /* Determine if this channel has fewer calls than previous channels. */
          if (channel_ptr->ch_calls_sch < least_calls)
          {
            least_calls_index = channel_index;
            least_calls = channel_ptr->ch_calls_sch;
85        }
        }
      }

      /* Found the channel that can support the call and has the least calls scheduled. */
90    if (least_calls_index >= 0)
      {
        /* Remember, the list position count is zero based. */
        if (LTRACE_CALL_SCH_ACTIVE)
        {
95        printf("In badd_call_scheduler.call scheduler, least calls index = %d.\n", least_calls_index);
          printf("In badd_call_scheduler.call scheduler, least calls = %d.\n", least_calls);
        }

        /* Time-stamp the request with the current time. */
100     op_ici_attr_set (call_iciptr, "time queued", op_sim_time());

        /* Put the call at the tail of the correct channel calls_scheduled list. */
        channel_listptr = (List*) op_prg_list_access(calls_scheduled, least_calls_index);
        op_prg_list_insert(channel_listptr, call_iciptr, OPC_LISTPOS_TAIL);
105
        /* If this is the first call on the channel, start the channel. */
        if (least_calls < 0)
        {
          if (LTRACE_CALL_SCH_ACTIVE)
110       {
            printf("In badd_call_scheduler.call scheduler, calling start call for channel %d.\n", least_calls_
            printf("In badd_call_scheduler.call scheduler, least calls = %d.\n", least_calls);
          }
          channel_iciptr = op_ici_create("badd_channel_ici");
115       op_ici_install(channel_iciptr);
          op_ici_attr_set (channel_iciptr, "channel number", least_calls_index);
          op_intrpt_schedule_self (op_sim_time (), BADD_CALL_SCH_START);
        }

120     /* Update the calls scheduled counter */
        least_calls = least_calls + 1;
        channel_ptr = (Badd_Channel_Desc*) op_prg_list_access(static_channels, least_calls_index);
        channel_ptr->ch_calls_sch = least_calls;

125     /* Update the channel completion timer. */
        if ((op_ici_attr_get (call_iciptr, "call duration", &call_duration) == OPC_COMPCODE_FAILURE) ||
          (op_ici_attr_get (call_iciptr, "call wait time", &call_wait_time) == OPC_COMPCODE_FAILURE))
          badd_call_sch_error("In badd_sch.call_sch: unable to get values from call_req iciptr.", OPC_NIL, OP
```

```
    if (op_sim_time() > channel_ptr->ch_compl_time)
130 {
/*      channel_ptr->ch_compl_time = op_sim_time() + call_duration + call_wait_time + channel_delay */
      channel_ptr->ch_compl_time = op_sim_time() + call_duration + channel_delay
                  + trans_delay;
/*      channel_ptr->ch_compl_time = op_sim_time() + call_duration; */
135 }
    else
    {
/*      channel_ptr->ch_compl_time = channel_ptr->ch_compl_time + call_duration + call_wait_time  */
      channel_ptr->ch_compl_time = channel_ptr->ch_compl_time + call_duration
140               + channel_delay + trans_delay;
/*      channel_ptr->ch_compl_time = channel_ptr->ch_compl_time + call_duration; */
    }

    if (LTRACE_CALL_TIMER_ACTIVE)
145   printf("In badd_call_scheduler.call_schedule: compl_time = %f.\n", channel_ptr->ch_compl_time);

    /* Signal this call was successfully scheduled. */
    if (LTRACE_CALL_SCH_ACTIVE)
    {
150   printf("In badd_call_scheduler.call scheduler: call scheduled.\n");
    }
    channel_scheduled = OPC_COMPCODE_SUCCESS;
  }

155 if (channel_scheduled == OPC_COMPCODE_FAILURE)
  {
    printf("Call parameters exceeds the capacity of all channels.\n");
  }
}
160
/* Turn the interrupts back on. */
op_intrpt_enable(OPC_INTRPT_SELF, BADD_CALL_SCHEDULER);
```

transition **call schedule -> dispatch**			
attribute	*value*	*type*	*default value*
name	tr_143	string	tr
condition		string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

forced state **reschedule**			
attribute	*value*	*type*	*default value*
name	reschedule	string	st
enter execs	(See below.)	textlist	
exit execs	(empty)	textlist	(empty)
status	forced	toggle	unforced

enter execs **reschedule**

```
req_iciptr = op_intrpt_ici();

if (req_iciptr == OPC_NIL)
   badd_call_sch_error("Unable to get call_req iciptr.", OPC_NIL, OPC_NIL);

if ((op_ici_attr_get (req_iciptr, "interarrival time", &int_arr_time) == OPC_COMPCODE_FAILURE) ||
   (op_ici_attr_get (req_iciptr, "packet size", &packet_size) == OPC_COMPCODE_FAILURE)     ||
   (op_ici_attr_get (req_iciptr, "call wait time", &call_wait_time) == OPC_COMPCODE_FAILURE) ||
   (op_ici_attr_get (req_iciptr, "call duration", &call_duration) == OPC_COMPCODE_FAILURE)  ||
   (op_ici_attr_get (req_iciptr, "dest addr", &dest_addr) == OPC_COMPCODE_FAILURE)          ||
   (op_ici_attr_get (req_iciptr, "QoS class", &qos_class) == OPC_COMPCODE_FAILURE)          ||
   (op_ici_attr_get (req_iciptr, "AAL type", &AAL_type) == OPC_COMPCODE_FAILURE)            ||
   (op_ici_attr_get (req_iciptr, "peak cell rate", &peak_cell_rate) == OPC_COMPCODE_FAILURE) ||
   (op_ici_attr_get (req_iciptr, "channel assigned", &call_release_channel) == OPC_COMPCODE_FAILURE))
   badd_call_sch_error("In badd_call_sch.reschedule: unable to get values from call_req iciptr", OPC_NIL,

op_ici_destroy(req_iciptr);

if (LTRACE_CALL_RESCHEDULER_ACTIVE)
{
   printf("In badd_call_scheduler.reschedule: int_arr_time = %f.\n", int_arr_time);
   printf("In badd_call_scheduler.reschedule: packet_size = %d.\n", packet_size);
   printf("In badd_call_scheduler.reschedule: call_wait_time = %f.\n", call_wait_time);
   printf("In badd_call_scheduler.reschedule: call_duration = %f.\n", call_duration);
   printf("In badd_call_scheduler.reschedule: dest_addr = %d.\n", dest_addr);
   printf("In badd_call_scheduler.reschedule: qos_class = %d.\n", qos_class);
   printf("In badd_call_scheduler.reschedule: AAL_type = %d.\n", AAL_type);
   printf("In badd_call_scheduler.reschedule: peak_cell_rate = %f.\n", peak_cell_rate);
   printf("In badd_call_scheduler.reschedule: req_iciptr = %x.\n", req_iciptr);
} /* End if (LTRACE_CALL_SCHEDULER_ACTIVE) */

/* Reduce the channel completion time for this call, it is added back in when the call */
/* is rescheduled.                                                                     */
channel_ptr = (Badd_Channel_Desc*) op_prg_list_access(static_channels, call_release_channel);
channel_ptr->ch_compl_time = channel_ptr->ch_compl_time - call_duration - channel_delay - trans_delay;

/* Create and set the fields in the interface ICI.                                     */
if_iciptr = op_ici_create ("badd_call_req_if_ici");

op_ici_attr_set (if_iciptr, "interarrival time", int_arr_time);
op_ici_attr_set (if_iciptr, "packet size", packet_size);
op_ici_attr_set (if_iciptr, "call wait time", call_wait_time);
op_ici_attr_set (if_iciptr, "call duration", call_duration);
op_ici_attr_set (if_iciptr, "dest addr", dest_addr);
op_ici_attr_set (if_iciptr, "QoS class", qos_class);
op_ici_attr_set (if_iciptr, "AAL type", AAL_type);
op_ici_attr_set (if_iciptr, "peak cell rate", peak_cell_rate);

op_prg_list_insert(calls_pending, if_iciptr, OPC_LISTPOS_HEAD);


/*if (sch_requested == OPC_COMPCODE_FAILURE)
/*    op_intrpt_schedule_self (op_sim_time (), BADD_CALL_SCHEDULER);
/*  */
```

...

...

transition **reschedule -> dispatch**

attribute	*value*	*type*	*default value*
name	tr_149	string	tr
condition		string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

forced state **check channel**

attribute	*value*	*type*	*default value*
name	check channel	string	st
enter execs	(See below.)	textlist	
exit execs	(empty)	textlist	(empty)
status	forced	toggle	unforced

enter execs **check channel**

```
/* Get the Ici passed from the call_generator and determine what channel must    */
/* be checked for additional calls. If the channel has additional calls waiting, */
/* then send an interrupt to start the next call.                                */

check_channel_iciptr = op_intrpt_ici();

if (op_ici_attr_get (check_channel_iciptr, "channel number", &check_chan_index) == OPC_COMPCODE_FAILURE)
  badd_call_sch_error("Unable to read check channel iciptr.", OPC_NIL, OPC_NIL);

op_ici_destroy(check_channel_iciptr);

if (LTRACE_CALL_SCH_ACTIVE)
  printf("In badd_call_scheduler.call scheduler.check_channel: channel = %d.\n", check_chan_index);

channel_ptr = (Badd_Channel_Desc*) op_prg_list_access(static_channels, check_chan_index);

/* Check the channel for additional calls waiting and start the next call if available. */
if (channel_ptr->ch_calls_sch >= 0)
{
  channel_iciptr = op_ici_create("badd_channel_ici");
  op_ici_install(channel_iciptr);
  op_ici_attr_set (channel_iciptr, "channel number", check_chan_index);
  op_intrpt_schedule_self (op_sim_time () + channel_delay, BADD_CALL_SCH_START);
}

if (LTRACE_CALL_TIMER_ACTIVE)
  printf("In badd_call_sch.check channel: current time = %f.\n", op_sim_time());
```

transition **check channel -> dispatch**

attribute	*value*	*type*	*default value*
name	tr_152	string	tr

condition		string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

APPENDIX F.
BADD_CALL_SCHEDULER_GREEDY.

...
...

External File Set

attribute	*value*	*type*	*default value*
external file set	badd_functions	typed file	

Process Model Comments

General Process Description:

The badd_call_scheduler schedules and manages all static channel assignments for the BADD network. This model schedules calls by assigning new calls to the channel based on a greedy min-min algorithm.

ICI Interfaces:

---

badd_call_req_if_ici
badd_call_gen_if_ici

Packet Formats:

None.

Statistic Interfaces:

None

Process Registry:

Published Attributes and Expected Attributes: None

Restrictions:

None

Process Model Attributes

Attribute **dest addr** properties

Property	*Value*	
Default Value:	NONE	
Data Type:	integer	
Attribute Description:	Private	
Auto. assign value:	FALSE	
Comments:	Specifies the destination of the call.	
Symbol Map:	*Symbol*	*Value*
	NONE	-1
Allow other values:	YES	

Attribute **scheduler delay** properties

Property	*Value*
Default Value:	0.0

Data Type:	double
Attribute Description:	Private
Auto. assign value:	FALSE
Units:	seconds
Comments:	The maximum time in seconds between scheduler events.

Attribute **transmission delay** properties

Property	*Value*
Default Value:	0.25
Data Type:	double
Attribute Description:	Private
Auto. assign value:	FALSE
Units:	seconds

Attribute **channel delay** properties

Property	*Value*
Default Value:	7.681E-05
Data Type:	double
Attribute Description:	Private
Auto. assign value:	FALSE
Units:	seconds
Comments:	The delay between calls on the same channel.

Attribute **calls pending** properties

Property	*Value*
Default Value:	0
Data Type:	integer
Attribute Description:	Private
Auto. assign value:	FALSE
Comments:	The maximum number of calls to store in the calls_pending list before running a schedule process.

Process Model Interface Attributes

Attribute **begsim intrpt** properties

Property	*Value*	*Inherit*
Assign Status:	hidden	
Initial Value:	enabled	N/A
Default Value:	disabled	YES
Data Type:	toggle	N/A
Attribute Description:	Private	N/A
Comments:	This attribute specifies whether a 'begin simulation interrupt' is generated for a processor module's root process at the start of the simulation.	YES

Symbol Map:	NONE	YES

Attribute **endsim intrpt** properties		
Property	*Value*	*Inherit*
Assign Status:	hidden	
Initial Value:	enabled	N/A
Default Value:	disabled	YES
Data Type:	toggle	N/A
Attribute Description:	Private	N/A
Comments:		YES
	This attribute specifies whether an 'end simulation interrupt' is generated for a processor module's root process at the end of the simulation.	
Symbol Map:	NONE	YES

Attribute **failure intrpts** properties		
Property	*Value*	*Inherit*
Assign Status:	hidden	
Initial Value:	disabled	N/A
Default Value:	disabled	YES
Data Type:	enumerated	N/A
Attribute Description:	Private	N/A
Comments:		YES
	This attribute specifies whether failure interrupts are generated for a processor module's root process upon failure of nodes or links in the network model.	
Symbol Map:	NONE	YES

Attribute **intrpt interval** properties		
Property	*Value*	*Inherit*
Assign Status:	hidden	
Initial Value:	disabled	N/A
Default Value:	disabled	YES
Data Type:	toggle double	N/A
Attribute Description:	Private	N/A
Units:	sec.	YES
Comments:		YES
	This attribute specifies how often regular interrupts are scheduled for the root process of a processor module.	
Symbol Map:	NONE	YES

Attribute **priority** properties		
Property	*Value*	*Inherit*
Assign Status:	hidden	
Initial Value:	0	N/A
Default Value:	0	YES
Data Type:	integer	N/A
Attribute Description:	Private	N/A
Low Range:	-32767 inclusive	YES
High Range:	32767 inclusive	YES

Comments:	This attribute is used to determine the execution order of events that are scheduled to occur at the same simulation time.	YES
Symbol Map:	NONE	YES

Attribute **recovery intrpts** properties

Property	*Value*	*Inherit*
Assign Status:	hidden	
Initial Value:	disabled	N/A
Default Value:	disabled	YES
Data Type:	enumerated	N/A
Attribute Description:	Private	N/A
Comments:	This attribute specifies whether recovery interrupts are scheduled for the processor module's root process upon recovery of nodes or links in the network model.	YES
Symbol Map:	NONE	YES

Attribute **super priority** properties

Property	*Value*	*Inherit*
Assign Status:	hidden	
Initial Value:	disabled	N/A
Default Value:	disabled	YES
Data Type:	toggle	N/A
Attribute Description:	Private	N/A
Comments:	This attribute is used to determine the execution order of events that are scheduled to occur at the same simulation time.	YES
Symbol Map:	NONE	YES

Process Model Simulation Attributes

Attribute **class_A_CDV_tolerance** properties

Property	*Value*
Default Value:	0.0
Data Type:	double
Attribute Description:	Private
Auto. assign value:	FALSE
Comments:	Upper bound on cell delay variance that two consecutive Quality of Service (QoS) class A cells may experience.

Attribute **class_B_CDV_tolerance** properties

Property	*Value*
Default Value:	0.0

Data Type:	double
Attribute Description:	Private
Auto. assign value:	FALSE
Comments:	Upper bound on cell delay variance that two consecutive Quality of Service (QoS) class B cells may experience.

Attribute **class_C_CDV_tolerance** properties

Property	*Value*
Default Value:	0.0
Data Type:	double
Attribute Description:	Private
Auto. assign value:	FALSE
Comments:	Upper bound on cell delay variance that two consecutive Quality of Service (QoS) class C cells may experience.

Attribute **class_D_CDV_tolerance** properties

Property	*Value*
Default Value:	0.0
Data Type:	double
Attribute Description:	Private
Auto. assign value:	FALSE
Comments:	Upper bound on cell delay variance that two consecutive Quality of Service (QoS) class D cells may experience.

Header Block

```
#include "/usr/local/mil3_dir/3.0.A_DL5/models/std/atm/ams_interfaces.h"
#include "/usr/local/mil3_dir/3.0.A_DL5/models/std/atm/ams_aal_interfaces.h"
#include "/usr/work/benton/op_code/badd_interface.h"

/* These are transition conditions. */
#define SIGNAL                  ((op_intrpt_type () == OPC_INTRPT_REMOTE) &&    \
                                (op_intrpt_code () == AMSC_INTERFACE_SIGNAL))

#define CALL_REQ_SIGNAL         ((op_intrpt_type () == OPC_INTRPT_REMOTE) &&    \
                                (op_intrpt_code () == BADD_REQUEST_SIGNAL))

#define AAL_CALL_SETUP          ((SIGNAL) &&                                    \
                                ((primitive == AMSC_AAL_ESTAB_Req) ||           \
                                (primitive == AMSC_AA_ESTAB_Ind)))

#define SAAL_SIGNAL             ((op_intrpt_type () == OPC_INTRPT_REMOTE) &&    \
                                (op_intrpt_code () == AMSC_SAAL_SIGNAL))
```

```
#define AAL_DATA                    (op_intrpt_type () == OPC_INTRPT_STRM)

#define EST_CON                     (SIGNAL && (primitive == AMSC_AAL_ESTAB_Con))

#define REL_IND                     (SIGNAL && (primitive == AMSC_AAL_RELEASE_Ind))

#define REL_CON                     (SIGNAL && (primitive == AMSC_AAL_RELEASE_Con))

#define CALL_START                  ((op_intrpt_type () == OPC_INTRPT_SELF) &&          \
                                    (op_intrpt_code () == BADD_CALL_SCH_START))

#define CALL_END                    ((op_intrpt_type () == OPC_INTRPT_SELF) &&          \
                                    (op_intrpt_code () == AMSC_TGEN_CALL_END))

#define SCHEDULER                   ((op_intrpt_type () == OPC_INTRPT_SELF) &&          \
                                    (op_intrpt_code () == BADD_CALL_SCHEDULER))

#define RESCHEDULE                  ((op_intrpt_type () == OPC_INTRPT_REMOTE) &&        \
                                    (op_intrpt_code () == BADD_CALL_RESCHEDULE))

#define CHECK_CHANNEL_SIGNAL        ((op_intrpt_type () == OPC_INTRPT_REMOTE) &&        \
                                    (op_intrpt_code () == BADD_CHECK_CHANNEL))

#define WAIT_CALL_DURATION          ((op_intrpt_type () == OPC_INTRPT_SELF) &&          \
                                    (op_intrpt_code () == AMSC_TGEN_WAIT_CALL_DURATION))

#define  END_SIMULATION             ((op_intrpt_type () == OPC_INTRPT_ENDSIM))

#define NEIGHBOR_NOTIFY             ((op_intrpt_type () == OPC_INTRPT_REMOTE) &&        \
                                    (op_intrpt_code () == AMSC_NEIGHBOR_NOTIFY))

#define NOTIFY_COMPLETE             (ams_neighbor_notify_is_complete (nbr_data_ptr) == OPC_TRUE)

/* These are self interrupt codes. */
#define   BADD_CALL_SCH_START               0
#define AMSC_TGEN_CALL_END                  1
#define AMSC_TGEN_DATA_GEN                  2
#define AMSC_TGEN_WAIT_CALL_DURATION        3
#define BADD_CALL_SCHEDULER                 4
#define BADD_CALL_RESCHEDULE                5
#define BADD_CHECK_CHANNEL                  6

/*  These are constants  */
#define MAX_CHANNELS                        1024
#define MAXSIZE                             11
#define MAX_COMPL_TIME                      9999999.9

/* The scheduler process will output trace information if these conditional are true. */
#define LTRACE_ACTIVE                       (debug_mode &&                              \
                                            (op_prg_odb_ltrace_active ("ams") ||        \
                                            op_prg_odb_ltrace_active ("ams_traf") ||    \
                                            op_prg_odb_ltrace_active ("ams_traf_gen")))

#define LTRACE_CONNECT_ACTIVE               (op_prg_odb_ltrace_active ("connect"))

#define LTRACE_CALL_SCHEDULER_ACTIVE        (op_prg_odb_ltrace_active ("call_scheduler"))

#define LTRACE_CALL_SCH_ACTIVE              (op_prg_odb_ltrace_active ("call_sch"))
```

```
#define LTRACE_CALL_DISP_ACTIVE        (op_prg_odb_ltrace_active ("call_disp"))
#define LTRACE_CALL_CHANNELS_ACTIVE    (op_prg_odb_ltrace_active ("call_channels"))
#define LTRACE_CALL_TIMER_ACTIVE       (op_prg_odb_ltrace_active ("call_timer"))
#define LTRACE_CALL_GREEDY_ACTIVE      (op_prg_odb_ltrace_active ("call_greedy"))

#define LTRACE_CALL_RESCHEDULER_ACTIVE     (op_prg_odb_ltrace_active ("call_rescheduler"))

/* Function declarations. */
void badd_call_sch_nbr_intrpt_proc ();
void badd_call_sch_spurious_signal_handle ();
void badd_call_sch_list_print ();
void badd_call_sch_matrix_print ();
void badd_call_sch_error ();
```

State Variable Block

```
int                 \debug_mode;
int                 \packet_size;
int                 \dest_addr;
int                 \AAL_type;
int                 \qos_class;
int                 \to_aal_stream_index;
int                 \to_req_stream_index;
int                 \sch_channel_count;
int                 \number_calls_pending;
int                 \max_calls_pending;
double              \avg_rate;
double              \channel_delay;
double              \trans_delay;
double              \time_to_next_scheduler;
double              \end_sim_time;
char                \pid_string [128];
Objid               \my_id;
Objid               \aal_module_id;
Objid               \req_module_id;
List*               \calls_pending;
List*               \calls_scheduled;
List*               \static_channels;
FILE*               \output_file;
FILE*               \output_calls;
Ici*                \aal_handle_iciptr;
Evhandle            \next_packet_arrival;
Evhandle            \call_end_intrpt;
Distribution*       \int_arrival_distptr;
Distribution*       \call_wait_distptr;
Distribution*       \call_duration_distptr;
AmsT_Traf_Contract* \traf_con_ptr;
AmsT_Neighbor_Data* \nbr_data_ptr;
Badd_Sch_Mod_Data   \Scheduler_Module;
Badd_Channel_Desc*  \channel_ptr;
```

Temporary Variable Block

```
    int             primitive;
    int             cd_packet_size;
    int             clark_dest_addr;
    int             calls_list_size;
 5  int             index = 0;
    int             channel_index;
    int             sch_channel_index;
    int             check_chan_index;
    int             call_bit_rate;
10  int             call_compl_channel;
    int             call_release_channel;
    int             total_channels = 0;
    int             channel_count = 0;
    int             value;
15  int             least_channel_index;
    int             least_call_index;
    int             temp_intrpt_type;
    int             temp_intrpt_code;
    int             remain_to_schedule;
20  int             row_index;
    int             col_index;
    int*            int_ptr;
    int*            channel_size_ptr;
    double          peak_cell_rate;
25  double          cdv_tolerance;
    double          int_arr_time;
    double          call_wait_time;
    double          call_duration;
    double          event_time;
30  double          least_compl_time;
    double          channel_compl_time;
    double          temp_double;
    double      temp_PCR;
    double          time_enqueued;
35  double          time_dequeued;
    double          time_in_queue;
    double*         compl_time_ptr;
    extern int      total_calls_requested;
    extern int      total_calls_generated;
40  extern int      total_calls_received;
    extern int      total_calls_active;
    extern int      max_calls_active;
    char            string[11];
    Ici*            if_iciptr;
45  Ici*            req_iciptr;
    Ici*            call_iciptr;
    Ici*            signal_iciptr;
    Ici*            setup_iciptr;
    Ici*            upper_handle_iciptr;
50  Ici*            check_channel_iciptr;
    Ici*            channel_iciptr;
    Ici*            sch_channel_iciptr;
    Prohandle       call_gen_prohandle;
    Prohandle*      call_gen_proptr;
55  Evhandle        this_event;
    Evhandle        next_event;
    Evhandle        scheduler_event;
```

```
List*                channel_listptr;
List*                sch_channel_listptr;
List*                temp_calls_sch_list;
List*                row_listptr;
FILE*                input_file;
Compcode             channel_scheduled;
Compcode             sch_requested;
AmsT_Traf_Contract*  tmp_traf_con_ptr;
```

Function Block

```
void
badd_call_sch_nbr_intrpt_proc (ndata_ptr, ndesc_ptr, state_ptr)
    AmsT_Neighbor_Data* ndata_ptr;
    AmsT_Neighbor_Desc* ndesc_ptr;
    void*               state_ptr;
    {
    AmsT_Neighbor_Verify_Desc   vdesc;
    int                         to_sw_stream_index;

    /** This procedure handles a neighbor notification event in an AMS          **/
    /** traf gen specific manner.  It determines the neighbor's object          **/
    /** ID and type, and verifies that there are a correct number of            **/
    /** interconnections.                                                       **/
    FIN (badd_call_sch_nbr_intrpt_proc (ndata_ptr, ndesc_ptr, state_ptr));

    /* Switch based on the AMS type.                                            */
    switch (ndesc_ptr->module_amstype)
        {
        case AMSC_MTYPE_AAL:
            {
            /* Build the verify desc data structure.                            */
            vdesc.mod_id            = my_id;
            vdesc.nbr_id            = ndesc_ptr->module_objid;
            vdesc.nbr_id_ptr        = &aal_module_id;
            vdesc.mod_name          = "BADD Call Scheduler";
            vdesc.nbr_name          = "AAL";
            vdesc.nbr_type          = AMSC_MTYPE_AAL;
            vdesc.from_nbr_strm_cnt         = 1;
            vdesc.from_nbr_strm_index_ptr   = OPC_NIL;
            vdesc.to_nbr_strm_cnt           = 1;
            vdesc.to_nbr_strm_index_ptr     = &to_aal_stream_index;

            /* Verify that the neighbor has the correct                         */
            /* characteristics.                                                 */
            ams_neighbor_verify (&vdesc);

            break;
            }

        case BADD_MTYPE_SCHEDULER_CLIENT:
            {
            /* Build the verify desc data structure.                            */
            vdesc.mod_id            = my_id;
            vdesc.nbr_id            = ndesc_ptr->module_objid;
            vdesc.nbr_id_ptr        = &req_module_id;
            vdesc.mod_name          = "BADD Call Scheduler";
```

```
                vdesc.nbr_name          = "call_req";
                vdesc.nbr_type          = BADD_MTYPE_SCHEDULER_CLIENT;
                vdesc.from_nbr_strm_cnt             = 1;
                vdesc.from_nbr_strm_index_ptr       = OPC_NIL;
                vdesc.to_nbr_strm_cnt               = 1;
                vdesc.to_nbr_strm_index_ptr         = &to_req_stream_index;

                /* Verify that the neighbor has the correct                    */
                /* characteristics.                                            */
                /* ams_neighbor_verify (&vdesc);              */

                break;
                }

            default:
                {
                /* This is an unexpected neighbor notification.                */
                /* Issue error message.                                        */
                op_sim_end ("Process received a neighbor notification from a module",
                    "of an unexpected type.  Simulation terminated.", OPC_NIL, OPC_NIL);

                break;
                }
            }

        FOUT;
        }

void
badd_call_sch_spurious_signal_handle ()
        {
        Ici*            if_iciptr;
        Ici*            release_if_iciptr;
        int             primitive;
        Ici*            ll_handle_iciptr;
        Packet*         pkptr;

        /* This procedure handles spurious interrupts.                         */
        /* Only three types of spurious interrupts can be accepted. All        */
        /* others must result in an op_sim_end (). These three                 */
        /* interrupts are:                                                     */
        /*   1. An AAL ESTAB Ind.                                              */
        /*   2. An AAL RELEASE Con.                                            */
        /*   3. A packet arrival.                                              */
        FIN (badd_call_sch_spurious_signal_handle ());

        if (AAL_CALL_SETUP)
            {
            /* This is a signal from the AAL.                                  */

            /* Obtain the ICI pointer.                                         */
            if_iciptr = op_intrpt_ici ();

            /* Obtain the primitive from the ICI.                              */
            op_ici_attr_get (if_iciptr, "primitive", &primitive);

            /* Obtain the 'lower layer handle' from the ICI.                   */
            op_ici_attr_get (if_iciptr, "lower layer handle", &ll_handle_iciptr);

```

```
        /* Switch on the 'primitive'.          */
        switch (primitive)
            {
            case AMSC_AAL_ESTAB_Ind:
110             {
                /* This is an 'establish indication' from the AAL.          */

                if (LTRACE_ACTIVE)
                    {
115                 op_prg_odb_print_major (pid_string, "Received spurious AAL ESTABLISH Request signal.",
                        "Call not accepted.", "Sending AAL RELEASE Request signal.", OPC_NIL);
                    }

                /* Set the 'upper layer handle' in the interface            */
120             /* ICI for the 'establish indication' signal, since         */
                /* the lower layer expects it to be filled in. The          */
                /* ICI is not destroyed, since this is a forced             */
                /* interrupt and the lower layer process expects            */
                /* to obtain the handle from the ICI when the               */
125             /* control returns.                                         */
                op_ici_attr_set (if_iciptr, "upper layer handle", OPC_NIL);

                /* Simply send an AAL_RELEASE_Req to request that           */
                /* the connection be released.                              */
130             release_if_iciptr = op_ici_create (AMSC_INTERFACE_ICI);
                op_ici_attr_set (release_if_iciptr, "primitive", AMSC_AAL_RELEASE_Req);
                op_ici_attr_set (release_if_iciptr, "lower layer handle", ll_handle_iciptr);
              op_ici_install (release_if_iciptr);

135             /* Send a remote interrupt which will carry the ICI to the AAL module. */
                op_intrpt_schedule_remote (op_sim_time (), AMSC_INTERFACE_SIGNAL, aal_module_id);

                break;
                }
140
            case AMSC_AAL_RELEASE_Con:
                {
                /* This is a 'release confirm' from the AAL.                */

145             if (LTRACE_ACTIVE)
                    {
                    op_prg_odb_print_major (pid_string, "Received spurious AAL RELEASE Confirm signal.",
                        "This in response to AAL RELEASE Request terminating spurious connection.",
                        OPC_NIL);
150                 }

                /* This is a response to our earlier 'release               */
                /* request' which was a response to the                     */
                /* spurious 'estab indication'.                             */
155             /* Do nothing other than destroy the ICI.                   */
                op_ici_destroy (if_iciptr);

                break;
                }
160
            default:
                {
                /* This is some other completely unexpected                 */
                /* signal. Issue error message and terminate                */
```

```
                /* simulation.                                          */
                op_sim_end ("In badd_call_scheduler, Received unexpected signal.", "", "", "");
                }
            }
        }
    else if (op_intrpt_type () == OPC_INTRPT_STRM)
        {
        /* This is a spurious packet arrival. The ams_traf_gen          */
        /* just destroys this packet.                                   */
        if (LTRACE_ACTIVE)
            {
            op_prg_odb_print_major (pid_string, "Received spurious DATA packet.",
                "Destroying the packet.", OPC_NIL);
            }

        /* Get the packet fromt the stream and destroy it.              */
        pkptr = op_pk_get (op_intrpt_strm ());
        op_pk_destroy (pkptr);
        }
    else
        {
        /* This is some other completely unexpected                     */
        /* interrupt. Issue error message and terminate                 */
        /* simulation.                                                  */
        op_sim_end ("Received unexpected interrupt.", "", "", "");
        }

    FOUT;
    }

void
badd_call_sch_list_print (calls_list)
  List*     calls_list;
  {
  int   list_size;
  int   index = 0;
  int   list_packet_size;
  int   list_dest_addr;
  int   list_qos_class;
  int   list_AAL_type;
  double list_bit_rate;
  double list_int_arr_time;
  double list_call_wait_time;
  double list_call_duration;
  double list_peak_cell_rate;
  Ici*  req_iciptr;

  /* This procedure will print all the calls_descs in the calls_pending list */
  FIN (badd_call_sch_list_print(calls_list));

  list_size = op_prg_list_size(calls_list);

  if (list_size == 0)
  {
    printf("      Attempting to print empty call_list.\n");
  }

  for (index = 0; index < list_size; index++)
  {
```

```
    req_iciptr = (Ici*) op_prg_list_access (calls_list, index);
225
    if (req_iciptr == OPC_NIL)
      badd_call_sch_error("Unable to get req_iciptr from calls_pending list.", OPC_NIL, OPC_NIL);

    if ((op_ici_attr_get (req_iciptr, "interarrival time", &list_int_arr_time) == OPC_COMPCODE_FAILURE) ||
230   (op_ici_attr_get (req_iciptr, "packet size", &list_packet_size) == OPC_COMPCODE_FAILURE)      ||
      (op_ici_attr_get (req_iciptr, "call wait time", &list_call_wait_time) == OPC_COMPCODE_FAILURE) ||
      (op_ici_attr_get (req_iciptr, "call duration", &list_call_duration) == OPC_COMPCODE_FAILURE)  ||
      (op_ici_attr_get (req_iciptr, "dest addr", &list_dest_addr) == OPC_COMPCODE_FAILURE)          ||
      (op_ici_attr_get (req_iciptr, "QoS class", &list_qos_class) == OPC_COMPCODE_FAILURE)          ||
235   (op_ici_attr_get (req_iciptr, "AAL type", &list_AAL_type) == OPC_COMPCODE_FAILURE)            ||
      (op_ici_attr_get (req_iciptr, "peak cell rate", &list_peak_cell_rate) == OPC_COMPCODE_FAILURE))
      badd_call_sch_error("Unable to get values from call_req iciptr", OPC_NIL, OPC_NIL);

    printf("In badd_call_scheduler.printing calls_pending list: int_arr_time = %f.\n",
240       list_int_arr_time);
    printf("In badd_call_scheduler.printing calls_pending list: packet_size = %d.\n",
        list_packet_size);
    printf("In badd_call_scheduler.printing calls_pending list: call_wait_time = %f.\n",
        list_call_wait_time);
245 printf("In badd_call_scheduler.printing calls_pending list: call_duration = %f.\n",
        list_call_duration);
    printf("In badd_call_scheduler.printing calls_pending list: dest_addr = %d.\n",
        list_dest_addr);
    printf("In badd_call_scheduler.printing calls_pending list: qos_class = %d.\n",
250       list_qos_class);
    printf("In badd_call_scheduler.printing calls_pending list: AAL_type = %d.\n",
        list_AAL_type);
    printf("In badd_call_scheduler.printing calls_pending list: peak_cell_rate = %f.\n",
        list_peak_cell_rate);
255 printf("In badd_call_scheduler.printing calls_pending list: bit_rate = %f.\n",
        list_peak_cell_rate * 424);
  } /* End for loop */

  FOUT;
260 }

void
badd_call_sch_matrix_print (sch_matrix)
  List*     sch_matrix;
265 {
  int    number_of_rows;
  int    number_of_cols;
  int    list_row_index;
  int    list_col_index;
270 List*  matrix_row_ptr;
  double* list_compl_time;

  /* This procedure will print all the elements in the call scheduling matrix */
  FIN (badd_call_sch_matrix_print(sch_matrix));
275

  number_of_rows = op_prg_list_size(sch_matrix);

  if (number_of_rows == 0)
  {
280   printf("      Attempting to print empty sch_matrix.\n");
  }
```

```
    for (list_row_index = 0; list_row_index < number_of_rows; list_row_index++)
    {
285   matrix_row_ptr = (List*) op_prg_list_access (sch_matrix, list_row_index);

      if (matrix_row_ptr == OPC_NIL)
        badd_call_sch_error("Unable to get row from sch_matrix.", OPC_NIL, OPC_NIL);

290   number_of_cols = op_prg_list_size(matrix_row_ptr);

      if (number_of_cols == 0)
      {
        printf("      Attempting to print empty row from sch_matrix.\n");
295   }

      for (list_col_index = 0; list_col_index < number_of_cols; list_col_index++)
      {

300     list_compl_time = (double*) op_prg_list_access (matrix_row_ptr, list_col_index);

        printf("In badd_call_scheduler.printing sch_matrix elements: row %d, col %d, compl_time %f.\n",
          list_row_index, list_col_index, *list_compl_time);
      } /* End for (list_col_index = 0; list_col_index < number_of_cols; list_col_index++) */
305
    } /* End for (list_row_index = 0; list_row_index < number_of_rows; list_row_index++) */

    FOUT;
    }
310
void
badd_call_sch_error (msg0, msg1, msg2)
    char*       msg0;
    char*       msg1;
315 char*       msg2;
    {
    /** Print an error message and exit the simulation. **/
    FIN (badd_call_sch_error (msg0, msg1, msg2));

320 op_sim_end ("Error in badd_call_scheduler process:",
        msg0, msg1, msg2);

    FOUT;
    }
325
```

Diagnostic Block

```
/* Display connection information, if requested. */
if (LTRACE_CONNECT_ACTIVE)
    {
5   /* Print information. */
    ams_neighbor_data_print (nbr_data_ptr, ams_neighbor_desc_print_noop);
    }
```

forced state **INIT**

attribute	*value*	*type*	*default value*
name	INIT	string	st
enter execs	(See below.)	textlist	
exit execs	(empty)	textlist	(empty)
status	forced	toggle	unforced

enter execs **INIT**

```
   /* Determine the initial values of the state variables and set up          */
   /* the initial state of this instantiation of the ams_traf_gen             */
   /* process.                                                                */

 5 /* Obtain the object ID of this process' parent module.                    */
   my_id = op_id_self ();

   /* Determine whether or not the simulation is in debug mode.               */
   debug_mode = op_sim_debug ();
10
   /* Initialize the AAL module object ID to NULL value.                      */
   aal_module_id = OPC_OBJID_NULL;
   req_module_id = OPC_OBJID_NULL;

15 /* Create a list for storing the calls as they arrive */
   calls_pending = op_prg_list_create();

   /* Read the static channels input file and create lists accordingly. */
   static_channels = op_prg_list_create();
20 calls_scheduled = op_prg_list_create();

   /* Open the channel definition file. */
   if ((input_file = fopen("/usr/work/benton/input_channels", "r"))
                                        == NULL)
25 {
     badd_call_sch_error("Problem opening static channel definition file.",
                                 OPC_NIL, OPC_NIL);
   }
   if (fgets ( string, MAXSIZE, input_file ) != NULL)
30 {
     if (LTRACE_CALL_CHANNELS_ACTIVE)
       printf("In badd_call_sch.init; string is %s.\n", string);
     if (get_digit(string, &value) == OPC_COMPCODE_FAILURE)
     {
35     badd_call_sch_error ("Invalid number of static channels.",
                  OPC_NIL, OPC_NIL);
     }
     total_channels = value;
     if (LTRACE_CALL_CHANNELS_ACTIVE)
40   {
       printf("In badd_call_sch.load_static_channels function after reading channel");
       printf(" count;total_channels = %d.\n", total_channels);
     }
   }
45 if (total_channels > MAX_CHANNELS)
   {
     badd_call_sch_error("Requested virtual channels exceeds maximum allowed.",
```

```
                OPC_NIL, OPC_NIL);
  }
50
  while ((fgets ( string, MAXSIZE, input_file ) != NULL) &&
      (channel_count < total_channels))
  {
    if (LTRACE_CALL_CHANNELS_ACTIVE)
55  {
      printf("In badd_call_sch.init, reading each channel size;");
      printf(" channel_count = %d.\n", channel_count);
      printf("In badd_call_sch.init; string value is %s.\n", string);
    }
60  if (get_digit(string, &value) == OPC_COMPCODE_FAILURE)
    {
      badd_call_sch_error ("Invalid number for static channels.",
                OPC_NIL, OPC_NIL);
    }
65  if (value > 8000000)
    {
      badd_call_sch_error (
    }
70  channel_ptr = (Badd_Channel_Desc*) op_prg_mem_alloc(sizeof(Badd_Channel_Desc));
    if (channel_ptr == OPC_NIL)
    {
      badd_call_sch_error("Unable to allocate memory for channel definition.",
                OPC_NIL, OPC_NIL);
75  }
    channel_ptr->ch_capacity = value;
    channel_ptr->ch_calls_sch = -1;
    channel_ptr->ch_calls_compl = 0;
    channel_ptr->ch_compl_time = 0.0;
80
    op_prg_list_insert(static_channels, channel_ptr, OPC_LISTPOS_TAIL);
    if (LTRACE_CALL_CHANNELS_ACTIVE)
    {
      printf("In badd_call_sch.init, channel value saved = %d.\n", channel_ptr->ch_capacity);
85  }

    /* Create a list to store the calls for this channel */
    channel_listptr = op_prg_list_create();
    if (channel_listptr == OPC_NIL)
90  {
      badd_call_sch_error("Unable to allocate memory for channel list.",
                OPC_NIL, OPC_NIL);
    }
    op_prg_list_insert(calls_scheduled, channel_listptr, OPC_LISTPOS_TAIL);
95  channel_count = channel_count + 1;
  }
  if (LTRACE_CALL_CHANNELS_ACTIVE)
  {
    channel_count = op_prg_list_size(static_channels);
100 printf("In badd_call_sch.init, elements in static channels ");
    printf("list = %d.\n", channel_count);
    channel_count = op_prg_list_size(calls_scheduled);
    printf("In badd_call_sch.init, elements in sch_channels ");
    printf("list = %d.\n", channel_count);
105 }

  fclose(input_file);
```

```
    /* Open file for storing call timer information. */
110 if ((output_file = fopen("/usr/work/benton/output_call_times", "w"))
                                        == NULL)
    {
      badd_call_sch_error("Problem opening output_call_times file.",
                                    OPC_NIL, OPC_NIL);
115 }

    /* Write a file format statement as the first line in the file. */
    fputs("File Format: Channel Enqueued_Time Dequeued_Time Time_In_Queue PCR\n",
        output_file);
120
    /* Open file for storing calls completed information. */
    if ((output_calls = fopen("/usr/work/benton/output_call_completed", "w"))
                                        == NULL)
    {
125   badd_call_sch_error("Problem opening output_call_completed file.",
                                    OPC_NIL, OPC_NIL);
    }

    /* Write a file format statement as the first line in the file. */
130 fputs("File Format: Channel Time_completed PCR.\n", output_calls);

    sch_channel_count = op_prg_list_size(static_channels);

    /* Generate PID display string.                                    */
135 sprintf (pid_string, "badd_call_scheduler PID (%d)", op_pro_id (op_pro_self ()));

    /* Obtain and send out neighbor information.                       */
    nbr_data_ptr = ams_neighbor_data_build ();
    ams_neighbor_notify (nbr_data_ptr, AMSC_MTYPE_AAL_CLIENT);
140
    if (LTRACE_CALL_SCHEDULER_ACTIVE)
    {
      printf("In badd_call_scheduler.init: print neighbor info.\n");
      ams_neighbor_data_print (nbr_data_ptr, ams_neighbor_desc_print_noop);
145 }

    /* This Scheduler_Module is attached to any process spawned by */
    /* this badd_call_requestor process.                           */
    /*Scheduler_Module = (Badd_Sch_Mod_Data*) op_prg_mem_alloc(sizeof(Badd_Sch_Mod_Data)); */
150 op_pro_modmem_install(&Scheduler_Module);

    /* Initialize the model parameter attributes. */
    op_ima_obj_attr_get (my_id, "scheduler delay", &time_to_next_scheduler);
    op_ima_obj_attr_get (my_id, "transmission delay", &trans_delay);
155 op_ima_obj_attr_get (my_id, "channel delay", &channel_delay);
    op_ima_obj_attr_get (my_id, "calls pending", &max_calls_pending);

    /* Initialize the end_sim_time varaible */
    op_ima_sim_attr_get(OPC_IMA_DOUBLE, "duration", &end_sim_time);
160
    if (LTRACE_CALL_SCHEDULER_ACTIVE)
    {

      printf("In badd_call_scheduler.init state: Leaving init state. \n");
165
    }
```

...
...

transition **INIT -> config**

attribute	*value*	*type*	*default value*
name	tr_1	string	tr
condition		string	
executive		string	
color	RGB333	color	RGB333
drawing style	line	toggle	spline

unforced state **config**

attribute	*value*	*type*	*default value*
name	config	string	st
enter execs	(empty)	textlist	(empty)
exit execs	(See below.)	textlist	
status	unforced	toggle	unforced

exit execs **config**

```
    /* Ams_traf_gen expects either a neighbor notification interrupt,          */
    /* or a spurious signal.                                                   */
    if (NEIGHBOR_NOTIFY)
        {
5       /* This is a 'neighbor notify' signal.                                 */
        if (LTRACE_ACTIVE)
            {
            op_prg_odb_print_major (pid_string, "Received neighbor notification.", OPC_NIL);
            }
10
        /* Handle the neighbor notification.                                   */
        ams_neighbor_interrupt_handle (nbr_data_ptr, badd_call_sch_nbr_intrpt_proc, OPC_NIL);
        }
    else
15      {
        /* This is a spurious interrupt. Handle appropriately.                 */
        badd_call_sch_spurious_signal_handle ();
        }
```

transition **config -> config**

attribute	*value*	*type*	*default value*
name	tr_0	string	tr
condition	default	string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

...
...

transition **config -> shared data**

attribute	*value*	*type*	*default value*
name	tr_118	string	tr
condition	NOTIFY_COMPLETE	string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

unforced state **dispatch**

attribute	*value*	*type*	*default value*
name	dispatch	string	st
enter execs	(See below.)	textlist	
exit execs	(See below.)	textlist	
status	unforced	toggle	unforced

enter execs **dispatch**

```
   /* Compare the number of calls in the calls_pending list. */
   /* If the number exceeds the max_calls_pending, schedule  */
   /* a call to execute the scheduler.                       */
   number_calls_pending = op_prg_list_size(calls_pending);
5
   if (LTRACE_CALL_GREEDY_ACTIVE)
     printf("In badd_call_scheduler.dispatch: calls pending = %d.\n", number_calls_pending);

   if (number_calls_pending > max_calls_pending)
10   op_intrpt_schedule_self (op_sim_time (), BADD_CALL_SCHEDULER);

   /* Schedule the next scheduling event */
   if (number_calls_pending > 0)
     op_intrpt_schedule_self (op_sim_time () + time_to_next_scheduler, BADD_CALL_SCHEDULER);
15
```

exit execs **dispatch**

```
   temp_intrpt_type = op_intrpt_type();
   temp_intrpt_code = op_intrpt_code();

   if (LTRACE_CALL_DISP_ACTIVE)
5  {
     printf("In badd_call_scheduler.dispatch: received SIGNAL.\n");
     printf("In badd_call_scheduler.dispatch: intrpt_type = %d.\n", temp_intrpt_type);
     printf("In badd_call_scheduler.dispatch: intrpt_code = %d.\n", temp_intrpt_code);
   } /* End if (LTRACE_CALL_DISP_ACTIVE) */
10
   if (SIGNAL)
   {
     signal_iciptr = op_intrpt_ici();
     if (signal_iciptr == OPC_NIL)
15     badd_call_sch_error("Unable to get signal_iciptr.", OPC_NIL, OPC_NIL);

     if (LTRACE_CALL_DISP_ACTIVE)
```

...
...

```
    {
      op_ici_print(signal_iciptr);
20  } /* End if (LTRACE_CALL_DISP_ACTIVE) */

    op_ici_attr_get(signal_iciptr, "primitive", &primitive);
}
```

transition **dispatch -> call request**

attribute	*value*	*type*	*default value*
name	tr_111	string	tr
condition	CALL_REQ_SIGNAL	string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

transition **dispatch -> send data**

attribute	*value*	*type*	*default value*
name	tr_131	string	tr
condition	EST_CON	string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

transition **dispatch -> call complete**

attribute	*value*	*type*	*default value*
name	tr_136	string	tr
condition	REL_CON	string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

transition **dispatch -> call released**

attribute	*value*	*type*	*default value*
name	tr_139	string	tr
condition	REL_IND	string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

transition **dispatch -> call schedule**

attribute	*value*	*type*	*default value*
name	tr_142	string	tr
condition	SCHEDULER	string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

...
...

transition **dispatch -> End Sim**			
attribute	*value*	*type*	*default value*
name	tr_145	string	tr
condition	END_SIMULATION	string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

transition **dispatch -> reschedule**			
attribute	*value*	*type*	*default value*
name	tr_148	string	tr
condition	RESCHEDULE	string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

transition **dispatch -> check channel**			
attribute	*value*	*type*	*default value*
name	tr_151	string	tr
condition	CHECK_CHANNEL_SIGNAL	string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

transition **dispatch -> start call**			
attribute	*value*	*type*	*default value*
name	tr_154	string	tr
condition	CALL_START	string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

transition **dispatch -> error**			
attribute	*value*	*type*	*default value*
name	tr_157	string	tr
condition	default	string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

...
...

forced state **start call**

attribute	*value*	*type*	*default value*
name	start call	string	st
enter execs	(See below.)	textlist	
exit execs	(empty)	textlist	(empty)
status	forced	toggle	unforced

enter execs **start call**

```
sch_channel_iciptr = op_intrpt_ici();
if ((op_ici_attr_get (sch_channel_iciptr, "channel number", &sch_channel_index) == OPC_COMPCODE_FAILURE))
  badd_call_sch_error("In start_call: unable to get values from sch_channel iciptr",
            OPC_NIL, OPC_NIL);

/* Destroy the ici to recover space, Garbage collection. */
op_ici_destroy(sch_channel_iciptr);

if (LTRACE_CALL_SCH_ACTIVE)
  printf("In call scheduler.start_call: sch_channel index = %d.\n", sch_channel_index);
sch_channel_listptr = (List*) op_prg_list_access(calls_scheduled, sch_channel_index);
call_iciptr = (Ici*) op_prg_list_remove(sch_channel_listptr, OPC_LISTPOS_HEAD);

if (call_iciptr == OPC_NIL)
  badd_call_sch_error("In start call, unable to get call_iciptr from calls_list.", OPC_NIL, OPC_NIL);

if ((op_ici_attr_get (call_iciptr, "interarrival time", &int_arr_time) == OPC_COMPCODE_FAILURE) ||
  (op_ici_attr_get (call_iciptr, "packet size", &packet_size) == OPC_COMPCODE_FAILURE)    ||
  (op_ici_attr_get (call_iciptr, "call wait time", &call_wait_time) == OPC_COMPCODE_FAILURE) ||
  (op_ici_attr_get (call_iciptr, "call duration", &call_duration) == OPC_COMPCODE_FAILURE)  ||
  (op_ici_attr_get (call_iciptr, "dest addr", &dest_addr) == OPC_COMPCODE_FAILURE)      ||
  (op_ici_attr_get (call_iciptr, "QoS class", &qos_class) == OPC_COMPCODE_FAILURE)      ||
  (op_ici_attr_get (call_iciptr, "AAL type", &AAL_type) == OPC_COMPCODE_FAILURE)       ||
  (op_ici_attr_get (call_iciptr, "peak cell rate", &peak_cell_rate) == OPC_COMPCODE_FAILURE) ||
  (op_ici_attr_get (call_iciptr, "time queued", &time_enqueued) == OPC_COMPCODE_FAILURE))
  badd_call_sch_error("Unable to get values from call_req iciptr", OPC_NIL, OPC_NIL);

if (LTRACE_CALL_SCHEDULER_ACTIVE)
{
  printf("In badd_call_scheduler.start_call: int_arr_time = %f.\n", int_arr_time);
  printf("In badd_call_scheduler.start_call: packet_size = %d.\n", packet_size);
  printf("In badd_call_scheduler.start_call: call_wait_time = %f.\n", call_wait_time);
  printf("In badd_call_scheduler.start_call: call_duration = %f.\n", call_duration);
  printf("In badd_call_scheduler.start_call: dest_addr = %d.\n", dest_addr);
  printf("In badd_call_scheduler.start_call: qos_class = %d.\n", qos_class);
  printf("In badd_call_scheduler.start_call: AAL_type = %d.\n", AAL_type);
  printf("In badd_call_scheduler.start_call: peak_cell_rate = %f.\n", peak_cell_rate);
} /* End if (LTRACE_CALL_SCHEDULER_ACTIVE) */

if (op_ici_attr_set (call_iciptr, "channel assigned", sch_channel_index) == OPC_COMPCODE_FAILURE)
  badd_call_sch_error("Unable to set sch_channel_index in call_iciptr", OPC_NIL, OPC_NIL);

time_dequeued = op_sim_time();
time_in_queue = time_dequeued - time_enqueued;

/* Write queue times to the call timer output file. */
fprintf(output_file, "%d %f %f %f %f\n", sch_channel_index, time_enqueued, time_dequeued,
    time_in_queue, peak_cell_rate);
```

```
50  /* Update program counter */
    total_calls_generated = total_calls_generated + 1;
    total_calls_active = total_calls_active + 1;
    if (total_calls_active > max_calls_active)
      max_calls_active = total_calls_active;
55
    if (LTRACE_CALL_SCHEDULER_ACTIVE)
    {
      printf("In badd_call_scheduler, start; starting call to dest %d.\n",
          dest_addr);
60  } /* if (LTRACE_CALL_SCHEDULER_ACTIVE) */

    /* Spawn a child process to generate the call data */
    call_gen_prohandle = op_pro_create("clark_badd_call_generator", OPC_NIL);
    if (op_pro_valid(call_gen_prohandle) == OPC_FALSE)
65  {
      badd_call_sch_error("Unable to create call generator process", OPC_NIL, OPC_NIL);
    }

    if (LTRACE_CALL_SCHEDULER_ACTIVE)
70  {
      printf("In badd_call_scheduler, start; invoking call generator.\n");
      printf("In badd_call_scheduler.start: call_iciptr = %x.\n", call_iciptr);
      printf("In badd_call_scheduler.start_call; md_aal_module_id = %d.\n", Scheduler_Module.md_aal_module_id);

75  } /* if (LTRACE_CALL_SCHEDULER_ACTIVE) */

    if (LTRACE_CALL_TIMER_ACTIVE)
      printf("In badd_call_sch.start call: current time = %f.\n", op_sim_time());

80  /* When invoking the process, pass in the call desc */
    if (op_pro_invoke(call_gen_prohandle, call_iciptr) == OPC_COMPCODE_FAILURE)
    {
      badd_call_sch_error("Unable to invoke call generator process", OPC_NIL, OPC_NIL);
    }
85
```

transition **start call -> dispatch**

attribute	*value*	*type*	*default value*
name	tr_155	string	tr
condition		string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

forced state **send data**

attribute	*value*	*type*	*default value*
name	send data	string	st
enter execs	(See below.)	textlist	
exit execs	(empty)	textlist	(empty)
status	forced	toggle	unforced

enter execs **send data**

```
signal_iciptr = op_intrpt_ici();

if (signal_iciptr == OPC_NIL)
  badd_call_sch_error("Unable to get signal_iciptr.", OPC_NIL, OPC_NIL);

if (LTRACE_CALL_SCHEDULER_ACTIVE)
{
  printf("In badd_call_scheduler.send data: restarting call_gen.\n");
  op_ici_print(signal_iciptr);

} /* End if (LTRACE_CALL_SCHEDULER_ACTIVE) */

if (op_ici_attr_get(signal_iciptr, "upper layer handle", &upper_handle_iciptr) == OPC_COMPCODE_FAILURE)
  badd_call_sch_error("Unable to get upper layer handle.", OPC_NIL, OPC_NIL);

if (op_ici_attr_get(upper_handle_iciptr, "call_gen_prohandle", &call_gen_proptr) == OPC_COMPCODE_FAILURE)
  badd_call_sch_error("Unable to get call_gen prohandle.", OPC_NIL, OPC_NIL);

if (LTRACE_CALL_TIMER_ACTIVE)
  printf("In badd_call_sch.send data: current time = %f.\n", op_sim_time());

if (op_pro_invoke(*call_gen_proptr, signal_iciptr) == OPC_COMPCODE_FAILURE)
{
  badd_call_sch_error("Unable to restart call_gen process", OPC_NIL, OPC_NIL);
}
```

transition **send data -> dispatch**

attribute	*value*	*type*	*default value*
name	tr_132	string	tr
condition		string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

forced state **call complete**

attribute	*value*	*type*	*default value*
name	call complete	string	st
enter execs	(See below.)	textlist	
exit execs	(empty)	textlist	(empty)
status	forced	toggle	unforced

enter execs **call complete**

```
if (signal_iciptr == OPC_NIL)
  badd_call_sch_error("Unable to get signal_iciptr.", OPC_NIL, OPC_NIL);
```

```
if (LTRACE_CALL_SCHEDULER_ACTIVE)
{
  printf("In badd_call_scheduler.call_complete: restarting call_gen.\n");
  op_ici_print(signal_iciptr);

} /* End if (LTRACE_CALL_SCHEDULER_ACTIVE) */

total_calls_received = total_calls_received + 1;
total_calls_active = total_calls_active - 1;

if (op_ici_attr_get(signal_iciptr, "upper layer handle", &upper_handle_iciptr) == OPC_COMPCODE_FAILURE)
  badd_call_sch_error("Unable to get upper layer handle.", OPC_NIL, OPC_NIL);

if (op_ici_attr_get(signal_iciptr, "traffic contract", &tmp_traf_con_ptr) == OPC_COMPCODE_FAILURE)
  badd_call_sch_error("Unable to get traffic contract.", OPC_NIL, OPC_NIL);

if (op_ici_attr_get(signal_iciptr, "badd_call_gen_channel_id", &call_compl_channel) == OPC_COMPCODE_FAILURE)
  badd_call_sch_error("Unable to get call_gen prohandle.", OPC_NIL, OPC_NIL);

/* Write completion time and PCR to the calls completed output file. */
temp_PCR = tmp_traf_con_ptr->calling_ctd.src_traf_desc.pcr;
fprintf(output_calls, "%d %f %f\n", call_compl_channel, op_sim_time(), temp_PCR);


channel_ptr = (Badd_Channel_Desc*) op_prg_list_access(static_channels, call_compl_channel);
channel_ptr->ch_calls_sch = channel_ptr->ch_calls_sch - 1;
channel_ptr->ch_calls_compl = channel_ptr->ch_calls_compl + 1;

if (op_ici_attr_get(upper_handle_iciptr, "call_gen_prohandle", &call_gen_proptr) == OPC_COMPCODE_FAILURE)
  badd_call_sch_error("Unable to get call_gen prohandle.", OPC_NIL, OPC_NIL);

if (LTRACE_CALL_TIMER_ACTIVE)
  printf("In badd_call_sch.call complete: current time = %f.\n", op_sim_time());

if (op_pro_invoke(*call_gen_proptr, signal_iciptr) == OPC_COMPCODE_FAILURE)
{
  badd_call_sch_error("Unable to restart call_gen process", OPC_NIL, OPC_NIL);
}
```

transition **call complete -> dispatch**			
attribute	*value*	*type*	*default value*
name	tr_137	string	tr
condition		string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

forced state **call released**			
attribute	*value*	*type*	*default value*
name	call released	string	st
enter execs	(See below.)	textlist	

exit execs	(empty)	textlist	(empty)
status	forced	toggle	unforced

enter execs **call released**

```
if (signal_iciptr == OPC_NIL)
  badd_call_sch_error("Unable to get signal_iciptr.", OPC_NIL, OPC_NIL);

if (LTRACE_CALL_SCHEDULER_ACTIVE)
{
  printf("In badd_call_scheduler.call_released: restarting call_gen.\n");
  op_ici_print(signal_iciptr);

} /* End if (LTRACE_CALL_SCHEDULER_ACTIVE) */

if (op_ici_attr_get(signal_iciptr, "upper layer handle", &upper_handle_iciptr) == OPC_COMPCODE_FAILURE)
  badd_call_sch_error("Unable to get upper layer handle.", OPC_NIL, OPC_NIL);

if (op_ici_attr_get(signal_iciptr, "badd_call_gen_channel_id", &call_release_channel) == OPC_COMPCODE_FAILURE)
  badd_call_sch_error("Unable to get call_gen call_release_channel.", OPC_NIL, OPC_NIL);

channel_ptr = (Badd_Channel_Desc*) op_prg_list_access(static_channels, call_release_channel);
channel_ptr->ch_calls_sch = channel_ptr->ch_calls_sch - 1;

if (op_ici_attr_get(upper_handle_iciptr, "call_gen_prohandle", &call_gen_proptr) == OPC_COMPCODE_FAILURE)
  badd_call_sch_error("Unable to get call_gen prohandle.", OPC_NIL, OPC_NIL);

total_calls_active = total_calls_active - 1;

if (op_pro_invoke(*call_gen_proptr, signal_iciptr) == OPC_COMPCODE_FAILURE)
{
  badd_call_sch_error("Unable to restart call_gen process", OPC_NIL, OPC_NIL);
}
```

transition **call released -> dispatch**

attribute	*value*	*type*	*default value*
name	tr_140	string	tr
condition		string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

forced state **shared data**

attribute	*value*	*type*	*default value*
name	shared data	string	st
enter execs	(See below.)	textlist	
exit execs	(empty)	textlist	(empty)
status	forced	toggle	unforced

...
...

enter execs **shared data**

```
   /* This Scheduler_Module is attached to any process spawned by */
   /* this badd_call_scheduler process.                           */
   op_pro_modmem_install(&Scheduler_Module);

5  /* Pass the neighbor information to all children processes.    */
   Scheduler_Module.md_neighbor_data_ptr = nbr_data_ptr;
   Scheduler_Module.md_aal_module_id = aal_module_id;
   Scheduler_Module.md_to_aal_stream_index = to_aal_stream_index;
   Scheduler_Module.md_calls_pending_ptr = calls_pending;
10 Scheduler_Module.md_channel_delay = channel_delay;
```

transition **shared data -> dispatch**

attribute	*value*	*type*	*default value*
name	tr_121	string	tr
condition		string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

unforced state **error**

attribute	*value*	*type*	*default value*
name	error	string	st
enter execs	(See below.)	textlist	
exit execs	(empty)	textlist	(empty)
status	unforced	toggle	unforced

enter execs **error**

```
   /* This is a spurious interrupt. Handle appropriately.        */

   printf("Bad signal received by badd_call_scheduler.\n");
   badd_call_sch_spurious_signal_handle ();
5
```

unforced state **End Sim**

attribute	*value*	*type*	*default value*
name	End Sim	string	st
enter execs	(See below.)	textlist	
exit execs	(See below.)	textlist	
status	unforced	toggle	unforced

...
...

enter execs **End Sim**

```
   /* Print Information during testing */
   printf("Calls currently active is %d.\n", total_calls_active);
   printf("Max calls active in this run was %d.\n", max_calls_active);

 5 calls_list_size = op_prg_list_size(calls_pending);

   printf("Calls remaining in the calls_pending list at termination = %d.\n", calls_list_size);

   printf("Calls remaining in the calls_scheduled lists at termination.\n");
10
   time_dequeued = end_sim_time;

   for (channel_index = 0; channel_index < sch_channel_count; channel_index++)
   {
15   channel_ptr = (Badd_Channel_Desc*) op_prg_list_access(static_channels, channel_index);
     calls_list_size = channel_ptr->ch_calls_sch;
     call_compl_channel = channel_ptr->ch_calls_compl;
     channel_compl_time = channel_ptr->ch_compl_time;
     printf("      Channel %d: calls compl = %d, calls remaining = %d, completion time = %f.\n",
20       channel_index, call_compl_channel, calls_list_size, channel_compl_time);
     channel_listptr = (List*) op_prg_list_access(calls_scheduled, channel_index);

     for (sch_channel_index = 0; sch_channel_index < calls_list_size; sch_channel_index++)
     {
25     call_iciptr = (Ici*) op_prg_list_access(channel_listptr, sch_channel_index);
       if ((op_ici_attr_get (call_iciptr, "time queued", &time_enqueued) == OPC_COMPCODE_FAILURE))
         badd_call_sch_error("In End Sim: unable to get values from call_req iciptr",
                     OPC_NIL, OPC_NIL);

30     time_in_queue = time_dequeued - time_enqueued;

       /* Write queue times to the call timer output file. */
       fprintf(output_file, "%d %f %f %f\n", channel_index, time_enqueued, 0.0, 0.0);
/*     fprintf(output_file, "%d %f %f %f\n", channel_index, time_enqueued, time_dequeued,
35 /*         time_in_queue); */
     }

     if (LTRACE_CALL_SCH_ACTIVE)
     {
40     badd_call_sch_list_print(channel_listptr);
     }
   }

   /* Close the Output files. */
45 fclose(output_file);
   fclose(output_calls);

   badd_call_sch_error("Ending simulation in badd_call_scheduler.End Sim.\n", OPC_NIL, OPC_NIL);
```

exit execs **End Sim**

...

...

forced state **call request**

attribute	*value*	*type*	*default value*
name	call request	string	st
enter execs	(See below.)	textlist	
exit execs	(empty)	textlist	(empty)
status	forced	toggle	unforced

enter execs **call request**

```
/* A call_request arrived from the call_requestor above. */
/* Must capture the call_request information from the    */
/* ICI and store the call in the call_pending_list       */

req_iciptr = op_intrpt_ici();

if (req_iciptr == OPC_NIL)
  badd_call_sch_error("Unable to get call_req iciptr.", OPC_NIL, OPC_NIL);

if ((op_ici_attr_get (req_iciptr, "interarrival time", &int_arr_time) == OPC_COMPCODE_FAILURE) ||
  (op_ici_attr_get (req_iciptr, "packet size", &packet_size) == OPC_COMPCODE_FAILURE)     ||
  (op_ici_attr_get (req_iciptr, "call wait time", &call_wait_time) == OPC_COMPCODE_FAILURE) ||
  (op_ici_attr_get (req_iciptr, "call duration", &call_duration) == OPC_COMPCODE_FAILURE)  ||
  (op_ici_attr_get (req_iciptr, "dest addr", &dest_addr) == OPC_COMPCODE_FAILURE)          ||
  (op_ici_attr_get (req_iciptr, "QoS class", &qos_class) == OPC_COMPCODE_FAILURE)          ||
  (op_ici_attr_get (req_iciptr, "AAL type", &AAL_type) == OPC_COMPCODE_FAILURE)            ||
  (op_ici_attr_get (req_iciptr, "peak cell rate", &peak_cell_rate) == OPC_COMPCODE_FAILURE))
  badd_call_sch_error("In badd_call_sch.call_req: unable to get values from call_req iciptr", OPC_NIL, O

/* Destroy the ici to recover space, garbage collection. */
op_ici_destroy(req_iciptr);

if (LTRACE_CALL_SCHEDULER_ACTIVE)
{
  printf("In badd_call_scheduler.call_request: int_arr_time = %f.\n", int_arr_time);
  printf("In badd_call_scheduler.call_request: packet_size = %d.\n", packet_size);
  printf("In badd_call_scheduler.call_request: call_wait_time = %f.\n", call_wait_time);
  printf("In badd_call_scheduler.call_request: call_duration = %f.\n", call_duration);
  printf("In badd_call_scheduler.call_request: dest_addr = %d.\n", dest_addr);
  printf("In badd_call_scheduler.call_request: qos_class = %d.\n", qos_class);
  printf("In badd_call_scheduler.call_request: AAL_type = %d.\n", AAL_type);
  printf("In badd_call_scheduler.call_request: peak_cell_rate = %f.\n", peak_cell_rate);
  printf("In badd_call_scheduler.call_request: req_iciptr = %x.\n", req_iciptr);
} /* End if (LTRACE_CALL_SCHEDULER_ACTIVE) */

/* Create and set the fields in the interface ICI.          */
/* -- Using local memory --                                */
if_iciptr = op_ici_create ("badd_call_req_if_ici");

op_ici_attr_set (if_iciptr, "interarrival time", int_arr_time);
op_ici_attr_set (if_iciptr, "packet size", packet_size);
op_ici_attr_set (if_iciptr, "call wait time", call_wait_time);
op_ici_attr_set (if_iciptr, "call duration", call_duration);
op_ici_attr_set (if_iciptr, "dest addr", dest_addr);
op_ici_attr_set (if_iciptr, "QoS class", qos_class);
op_ici_attr_set (if_iciptr, "AAL type", AAL_type);
op_ici_attr_set (if_iciptr, "peak cell rate", peak_cell_rate);
```

```
   op_prg_list_insert(calls_pending, if_iciptr, OPC_LISTPOS_TAIL);
50
   total_calls_requested = total_calls_requested + 1;

   /* Send an intrpt to start the scheduler if not requested. */
   sch_requested = OPC_COMPCODE_FAILURE;
55
   this_event = op_ev_current();
   next_event = op_ev_next_local(this_event);
   while (op_ev_valid(next_event))
   {
60   if ((op_ev_type(next_event) == OPC_INTRPT_SELF) &&
       (op_ev_code(next_event) == BADD_CALL_SCHEDULER))
     {
       if (LTRACE_CALL_SCH_ACTIVE)
         printf("Found scheduler event, stopping search.\n");
65     sch_requested = OPC_COMPCODE_SUCCESS;
       break;
     }
     else
     {
70     next_event = op_ev_next_local(next_event);
     }
   }

   /*if(sch_requested == OPC_COMPCODE_FAILURE)
75 /*{
   /*   if(LTRACE_CALL_SCH_ACTIVE)
   /*     printf("Sending for scheduler interrupt.\n");
   /*   op_intrpt_schedule_self(op_sim_time(), BADD_CALL_SCHEDULER);
   /*} */
```

transition **call request -> dispatch**			
attribute	*value*	*type*	*default value*
name	tr_113	string	tr
condition		string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

forced state **call schedule**			
attribute	*value*	*type*	*default value*
name	call schedule	string	st
enter execs	(See below.)	textlist	
exit execs	(empty)	textlist	(empty)
status	forced	toggle	unforced

enter execs **call schedule**

```
   /* Disable the intrpts to make this process atomic. */
   op_intrpt_disable(OPC_INTRPT_SELF, BADD_CALL_SCHEDULER, OPC_FALSE);
```

```
/* Remove all the other events of this type from the events list. */
this_event = op_ev_current();
next_event = op_ev_next_local(this_event);
while (op_ev_valid(next_event))
{
  if ((op_ev_type(next_event) == OPC_INTRPT_SELF) &&
    (op_ev_code(next_event) == BADD_CALL_SCHEDULER))
  {
    if (LTRACE_CALL_SCH_ACTIVE)
      printf("Found next sch event, deleting event.\n");
    scheduler_event = next_event;
    next_event = op_ev_next_local(scheduler_event);
    op_ev_cancel(scheduler_event);
  }
  else
    next_event = op_ev_next_local(next_event);
}

/* Determine the number of calls that must be scheduled. */
calls_list_size = op_prg_list_size(calls_pending);

if (LTRACE_CALL_GREEDY_ACTIVE)
  {
  if (calls_list_size == 0)
    printf("In badd_call_scheduler.call scheduler, attempted to schedule empty list.");
  else
    printf("In badd_call_scheduler.call scheduler, calls list size = %d.\n", calls_list_size);
  }

/* Initialize variables for find first call and channel to schedule. */
least_compl_time = MAX_COMPL_TIME;
least_channel_index = -1;
least_call_index = -1;

/* Create and Initialize the scheduling matrix. */
temp_calls_sch_list = op_prg_list_create();
for (row_index = 0; row_index < calls_list_size; row_index++)
{
  /* Access a call description in the calls_pending list. */
  call_iciptr = (Ici*) op_prg_list_access(calls_pending, row_index);

  if (call_iciptr == OPC_NIL)
    badd_call_sch_error("In badd_call_scheduler.call schedule, unable to get call_iciptr.",
                OPC_NIL, OPC_NIL);

  /* Get the call-duration, call_wait_time, and peak_cell_rate from the call descriptor. */
  if ((op_ici_attr_get (call_iciptr, "call duration", &call_duration) == OPC_COMPCODE_FAILURE) ||
    (op_ici_attr_get (call_iciptr, "call wait time", &call_wait_time) == OPC_COMPCODE_FAILURE) ||
    (op_ici_attr_get (call_iciptr, "peak cell rate", &peak_cell_rate) == OPC_COMPCODE_FAILURE))
    badd_call_sch_error("In badd_sch.call schedule: unable to get values from call_req iciptr.",
                OPC_NIL, OPC_NIL);

  if (LTRACE_CALL_GREEDY_ACTIVE)
  {
   printf("In badd_call_scheduler.call schedule: peak_cell_rate = %f.\n",
        peak_cell_rate);
  } /* End if (LTRACE_CALL_SCHEDULER_ACTIVE) */
```

```
    call_bit_rate = peak_cell_rate * AMSC_ATM_CELL_SIZE;

    /* Create channels list for this call. */
    sch_channel_listptr = op_prg_list_create();
    if (sch_channel_listptr == OPC_NIL)
      badd_call_sch_error("In badd_call_scheduler.call schedule, unable to create sch_channel_listptr.",
                  OPC_NIL, OPC_NIL);

    /* Insert the channels list into the scheduling matrix as a row. */
    op_prg_list_insert(temp_calls_sch_list, sch_channel_listptr, OPC_LISTPOS_TAIL);

    /* Calculate the completion time for this call on every channel and store in the scheduling */
    /* matrix.                                                                                  */
    for (col_index = 0; col_index < sch_channel_count; col_index++)
    {
      /* Allocate space to store channel completion time. */
      compl_time_ptr = (double*) op_prg_mem_alloc(sizeof(double));

      if (LTRACE_CALL_GREEDY_ACTIVE)
      {
        printf("In badd_call_scheduler.call scheduler: row_index = %d, col_index = %d.\n",
            row_index, col_index);
      }

      /* Get the channel descriptor. */
      channel_ptr = (Badd_Channel_Desc*) op_prg_list_access(static_channels, col_index);
      if (LTRACE_CALL_GREEDY_ACTIVE)
      {
        printf("In badd_call_scheduler.call scheduler, ch_capacity = %d.\n", channel_ptr->ch_capacity);
        printf("In badd_call_scheduler.call scheduler, bit rate = %d.\n", call_bit_rate);
      }

      /* Determine if this channel can support the call */
      if (call_bit_rate <= channel_ptr->ch_capacity)
      {
        if (op_sim_time () > channel_ptr->ch_compl_time)
        {
          *compl_time_ptr = op_sim_time() + call_duration + channel_delay + trans_delay;
/*          *compl_time_ptr = op_sim_time() + call_duration; */
        }
        else
        {
          *compl_time_ptr = channel_ptr->ch_compl_time + call_duration + channel_delay + trans_delay;
/*          *compl_time_ptr = channel_ptr->ch_compl_time + call_duration; */
        }

       /* Determine if this channel has a shorter completion time than all previous channels. */
        if (*compl_time_ptr < least_compl_time)
        {
          least_channel_index = col_index;
          least_call_index = row_index;
          least_compl_time = *compl_time_ptr;
        }
      }
      else
      {
        *compl_time_ptr = MAX_COMPL_TIME;
      }

```

```
        if (LTRACE_CALL_GREEDY_ACTIVE)
        {
          printf("In badd_call_scheduler.call schedule: compl time = %f.\n",
              *compl_time_ptr);
125     } /* End if (LTRACE_CALL_GREEDY_ACTIVE) */

        /* Store the completion time in the matrix. */
        op_prg_list_insert(sch_channel_listptr, compl_time_ptr, OPC_LISTPOS_TAIL);

130   } /* End for (col_index = 0; col_index < sch_channel_count; col_index++) */
    } /* for (row_index = 0; row_index < calls_list_size; row_index++) */

    /* Initialize the number of calls to schedule. */
    remain_to_schedule = calls_list_size;
135
    while (remain_to_schedule > 0)
    {
      /* During initialization of scheduling matrix, found the first call and channel to schedule. */
      /* On all subsequents calls, call and channel to schedule are completed at the end of the    */
140   /* while loop. */
      if (LTRACE_CALL_GREEDY_ACTIVE)
      {
        printf("In badd_call_scheduler.call schedule: least_channel %d, least_call %d, least_compl %f.\n"
            least_channel_index, least_call_index, least_compl_time);
145     badd_call_sch_matrix_print(temp_calls_sch_list);
      }

      if (least_call_index < 0)
        badd_call_sch_error("Unable to schedule any calls, call exceeds channel capacities.",
150                 OPC_NIL, OPC_NIL);

      /* Remove the call from the calls_pending list. */
      call_iciptr = (Ici*) op_prg_list_remove(calls_pending, least_call_index);

155   /* Time-stamp the request with the current time. */
      op_ici_attr_set (call_iciptr, "time queued", op_sim_time());

      /* Put the call at the tail of the correct channel calls_scheduled list. */
      channel_listptr = (List*) op_prg_list_access(calls_scheduled, least_channel_index);
160   op_prg_list_insert(channel_listptr, call_iciptr, OPC_LISTPOS_TAIL);

      /* Remove the row from the matrix and return the memory to system. This kernal */
      /* process also deallocates all memory for the list elements.                  */
      row_listptr = (List*) op_prg_list_remove(temp_calls_sch_list, least_call_index);
165   op_prg_list_free(row_listptr);

      /* Update the calls scheduled counter */
      channel_ptr = (Badd_Channel_Desc*) op_prg_list_access(static_channels, least_channel_index);
      channel_ptr->ch_calls_sch = channel_ptr->ch_calls_sch + 1;
170
      /* If this is the first call on the channel, start the channel. */
      if (channel_ptr->ch_calls_sch == 0)
      {
        if (LTRACE_CALL_GREEDY_ACTIVE)
175     {
          printf("In badd_call_scheduler.call scheduler, calling start call for channel %d.\n",
              least_channel_index);
        }
```

```
180     /* Identify the channel to start sending data. */
        channel_iciptr = op_ici_create("badd_channel_ici");
        op_ici_install(channel_iciptr);
        op_ici_attr_set (channel_iciptr, "channel number", least_channel_index);
        op_intrpt_schedule_self (op_sim_time (), BADD_CALL_SCH_START);
185   }

      /* Update the channel completion timer. */
      if ((op_ici_attr_get (call_iciptr, "call duration", &call_duration) == OPC_COMPCODE_FAILURE) ||
        (op_ici_attr_get (call_iciptr, "call wait time", &call_wait_time) == OPC_COMPCODE_FAILURE))
190     badd_call_sch_error("In badd_sch.call schedule: unable to get values from call_iciptr.",
                  OPC_NIL, OPC_NIL);

      if (op_sim_time () > channel_ptr->ch_compl_time)
      {
195     channel_ptr->ch_compl_time = op_sim_time() + call_duration + channel_delay + trans_delay;
/*      channel_ptr->ch_compl_time = op_sim_time() + call_duration; */
      }
      else
      {
200     channel_ptr->ch_compl_time = channel_ptr->ch_compl_time + call_duration + \
                    + channel_delay + trans_delay;
/*      channel_ptr->ch_compl_time = channel_ptr->ch_compl_time + call_duration; */
      }

205   if (LTRACE_CALL_GREEDY_ACTIVE)
      {
        printf("In badd_call_scheduler.call_schedule: compl_time = %f.\n", channel_ptr->ch_compl_time);
        printf("In badd_call_scheduler.call scheduler: call scheduled.\n");
      }
210   channel_scheduled = OPC_COMPCODE_SUCCESS;

      remain_to_schedule = remain_to_schedule - 1;

      col_index = least_channel_index;
215
      least_channel_index = -1;
      least_call_index = -1;
      least_compl_time = MAX_COMPL_TIME;

220   /* Get the channel completion time and update the channel column in the scheduling matrix. */
      channel_ptr = (Badd_Channel_Desc*) op_prg_list_access(static_channels, col_index);

      /* Step through each job in the calls_pending list and update the channel column in the */
      /* scheduling matrix with a new channel completion time.                                */
225   for (row_index = 0; row_index < remain_to_schedule; row_index++)
      {

        if (LTRACE_CALL_GREEDY_ACTIVE)
        {
230       printf("In call_scheduler.call_schedule: updating row %d, col %d.\n",
              row_index, col_index);
        }

        /* Get the location for the current matrix element. */
235     sch_channel_listptr = (List*) op_prg_list_access (temp_calls_sch_list, row_index);
        compl_time_ptr = (double*) op_prg_list_access (sch_channel_listptr, col_index);

        /* Only update the completion time if the call can run on this channel. */
```

```
        if (*compl_time_ptr < MAX_COMPL_TIME)
240     {
          /* Get the call_duration time from the call descriptor. */
          call_iciptr = (Ici*) op_prg_list_access(calls_pending, row_index);
          if ((op_ici_attr_get(call_iciptr, "call duration", &call_duration) == OPC_COMPCODE_FAILURE))
            badd_call_sch_error("In call_scheduler.schedule: unable to get call_duration.",
245                   OPC_NIL, OPC_NIL);

          if (op_sim_time () > channel_ptr->ch_compl_time)
          {
            *compl_time_ptr = op_sim_time() + call_duration + channel_delay + trans_delay;
250 /*          *compl_time_ptr = op_sim_time() + call_duration; */
          }
          else
          {
            *compl_time_ptr = channel_ptr->ch_compl_time + call_duration + channel_delay +
255                 trans_delay;
    /*        *compl_time_ptr = channel_ptr->ch_compl_time + call_duration; */
          }

        } /* End if (*compl_time_ptr < MAX_COMPL_TIME) */
260
      } /* End for (row_index = 0; row_index < remain_to_schedule; row_index++) */

      /* After updating the completion time for each call, step through the entire scheduling */
      /* matrix looking for the next call to schedule.                                        */
265   for (row_index = 0; row_index < remain_to_schedule; row_index++)
      {
        for (col_index = 0; col_index < sch_channel_count; col_index++)
        {
          /* Get the current location in the scheduling matrix. */
270       sch_channel_listptr = (List*) op_prg_list_access (temp_calls_sch_list, row_index);
          compl_time_ptr = (double*) op_prg_list_access (sch_channel_listptr, col_index);

          /* Determine if this channel has the shortest completion time. */
          if (*compl_time_ptr < least_compl_time)
275       {
            least_channel_index = col_index;
            least_call_index = row_index;
            least_compl_time = *compl_time_ptr;
          } /* End if (*compl_time_ptr < least_compl_time) */
280
        } /* End for (col_index = 0; col_index < sch_channel_count; col_index++) */

      } /* End for (row_index = 0; row_index < remain_to_schedule; row_index++) */

285 } /* End while (remain_to_schedule > 0) */

    /* remove the list to deallocate memory resources; garbage collection. */
    op_prg_list_free(temp_calls_sch_list);

290 /* Turn the interrupts back on. */
    op_intrpt_enable(OPC_INTRPT_SELF, BADD_CALL_SCHEDULER);
```

...
...

transition **call schedule -> dispatch**

attribute	*value*	*type*	*default value*
name	tr_143	string	tr
condition		string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

forced state **reschedule**

attribute	*value*	*type*	*default value*
name	reschedule	string	st
enter execs	(See below.)	textlist	
exit execs	(empty)	textlist	(empty)
status	forced	toggle	unforced

enter execs **reschedule**

```
req_iciptr = op_intrpt_ici();

if (req_iciptr == OPC_NIL)
   badd_call_sch_error("Unable to get call_req iciptr.", OPC_NIL, OPC_NIL);

if ((op_ici_attr_get (req_iciptr, "interarrival time", &int_arr_time) == OPC_COMPCODE_FAILURE) ||
   (op_ici_attr_get (req_iciptr, "packet size", &packet_size) == OPC_COMPCODE_FAILURE)      ||
   (op_ici_attr_get (req_iciptr, "call wait time", &call_wait_time) == OPC_COMPCODE_FAILURE) ||
   (op_ici_attr_get (req_iciptr, "call duration", &call_duration) == OPC_COMPCODE_FAILURE)   ||
   (op_ici_attr_get (req_iciptr, "dest addr", &dest_addr) == OPC_COMPCODE_FAILURE)          ||
   (op_ici_attr_get (req_iciptr, "QoS class", &qos_class) == OPC_COMPCODE_FAILURE)          ||
   (op_ici_attr_get (req_iciptr, "AAL type", &AAL_type) == OPC_COMPCODE_FAILURE)            ||
   (op_ici_attr_get (req_iciptr, "peak cell rate", &peak_cell_rate) == OPC_COMPCODE_FAILURE) ||
   (op_ici_attr_get (req_iciptr, "channel assigned", &call_release_channel) == OPC_COMPCODE_FAILURE))
   badd_call_sch_error("In badd_call_sch.reschedule: unable to get values from call_req iciptr", OPC_NIL,

op_ici_destroy(req_iciptr);

if (LTRACE_CALL_RESCHEDULER_ACTIVE)
{
   printf("In badd_call_scheduler.reschedule: int_arr_time = %f.\n", int_arr_time);
   printf("In badd_call_scheduler.reschedule: packet_size = %d.\n", packet_size);
   printf("In badd_call_scheduler.reschedule: call_wait_time = %f.\n", call_wait_time);
   printf("In badd_call_scheduler.reschedule: call_duration = %f.\n", call_duration);
   printf("In badd_call_scheduler.reschedule: dest_addr = %d.\n", dest_addr);
   printf("In badd_call_scheduler.reschedule: qos_class = %d.\n", qos_class);
   printf("In badd_call_scheduler.reschedule: AAL_type = %d.\n", AAL_type);
   printf("In badd_call_scheduler.reschedule: peak_cell_rate = %f.\n", peak_cell_rate);
   printf("In badd_call_scheduler.reschedule: req_iciptr = %x.\n", req_iciptr);

   printf("In badd_call_scheduler.reschedule: current sim time = %f.\n", op_sim_time());
} /* End if (LTRACE_CALL_RESCHEDULER_ACTIVE) */

/* Reduce the channel completion time for this call, it is added back in when the call */
/* is rescheduled.                                                                     */
channel_ptr = (Badd_Channel_Desc*) op_prg_list_access(static_channels, call_release_channel);
```

```
channel_ptr->ch_compl_time = channel_ptr->ch_compl_time - call_duration - channel_delay - trans_delay;

/* Create and set the fields in the interface ICI.                    */
if_iciptr = op_ici_create ("badd_call_req_if_ici");

op_ici_attr_set (if_iciptr, "interarrival time", int_arr_time);
op_ici_attr_set (if_iciptr, "packet size", packet_size);
op_ici_attr_set (if_iciptr, "call wait time", call_wait_time);
op_ici_attr_set (if_iciptr, "call duration", call_duration);
op_ici_attr_set (if_iciptr, "dest addr", dest_addr);
op_ici_attr_set (if_iciptr, "QoS class", qos_class);
op_ici_attr_set (if_iciptr, "AAL type", AAL_type);
op_ici_attr_set (if_iciptr, "peak cell rate", peak_cell_rate);

op_prg_list_insert(calls_pending, if_iciptr, OPC_LISTPOS_HEAD);

/* if(sch_requested == OPC_COMPCODE_FAILURE)
/*    op_intrpt_schedule_self(op_sim_time (), BADD_CALL_SCHEDULER); */
```

transition **reschedule -> dispatch**

attribute	*value*	*type*	*default value*
name	tr_149	string	tr
condition		string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

forced state **check channel**

attribute	*value*	*type*	*default value*
name	check channel	string	st
enter execs	(See below.)	textlist	
exit execs	(empty)	textlist	(empty)
status	forced	toggle	unforced

enter execs **check channel**

```
/* Get the Ici passed from the call_generator and determine what channel must    */
/* be checked for additional calls. If the channel has additional calls waiting, */
/* then send an interrupt to start the next call.                                */

check_channel_iciptr = op_intrpt_ici();

if (op_ici_attr_get (check_channel_iciptr, "channel number", &check_chan_index) == OPC_COMPCODE_FAILURE)
  badd_call_sch_error("Unable to read check channel iciptr.", OPC_NIL, OPC_NIL);

op_ici_destroy(check_channel_iciptr);

if (LTRACE_CALL_SCH_ACTIVE)
  printf("In badd_call_scheduler.call scheduler.check_channel: channel = %d.\n", check_chan_index);
```

```
15  channel_ptr = (Badd_Channel_Desc*) op_prg_list_access(static_channels, check_chan_index);

    /* Check the channel for additional calls waiting and start the next call if available. */
    if (channel_ptr->ch_calls_sch >= 0)
    {
20    channel_iciptr = op_ici_create("badd_channel_ici");
      op_ici_install(channel_iciptr);
      op_ici_attr_set (channel_iciptr, "channel number", check_chan_index);
      op_intrpt_schedule_self (op_sim_time () + channel_delay, BADD_CALL_SCH_START);
    }
25
    if (LTRACE_CALL_TIMER_ACTIVE)
      printf("In badd_call_sch.check channel: current time = %f.\n", op_sim_time());
```

transition **check channel -> dispatch**

attribute	*value*	*type*	*default value*
name	tr_152	string	tr
condition		string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

LIST OF REFERENCES

[1]Martin de Prycker. *Asynchronous Transfer Mode: solution for broadband ISDN, Third Edition.* Prentice Hall International (UK) Limited, London, 1995.

[2]Clifford J. Warner. Multiservice internet protocols for high performance networks. Briefing Conducted at NRAD, November 1996.

[3]William J. Neal. Badd technology transfer brief. Slide briefing presented by Dr Neal, March 1996.

[4]MIL 3, Inc, 3400 International Drive NW, Washington DC 20008. *OPNET Modeler Manual Series*, August 1996.

[5]Naval Command, Control, and Ocean Surveillance Center, Research, Development, Test and Evaluation Division, Code 422, 53140 Gatchell Road, San Diego, CA 92152-7400. *SmartNet Scheduling Tool v2.6 Users Guide*, June 1996.

[6]William Stallings. *Data and Computer Communications, Fourth Edition.* Prentice-Hall, Inc., Upper Saddle River, New Jersey, 1997.

[7]Andrew S. Tanenbaum. *Computer Networks, Third Edition.* Prentice-Hall, Inc., Upper Saddle River, New Jersey, 1996.

[8]Headquarters, Department of the Army. *Tactics, Techniques, and Procedures for the Tactical Internet, Coordinated Draft (Version 3)*, October 1996.

[9]Min Tan and Howard Jay Siegel. A stochastic model of a dedicated heterogeneous computing system for esatblishing a greedy approach to developing data relocation heuristics. In Debra Hensgen, editor, *Proceedings Sixth Heterogeneous Computing Workshop.* The IEEE Computer Society Office of Naval Research, 1997.

[10][SmartNet]/Heterogeneous Computing Team. BC2A/TACITUS/BADD INTEGRATION PLAN. Document Proposing Use of SmartNet Scheduling for JBS connectivity to BC2A., August 1996.

[11]Jeffrey Carpenter and Jim Galanis. Trip report from network testing and integration of GBS/BADD program in TFXXI. Trip report by two Electronics Engineers from US Army CECOM providing technical observations of the network in preparation for TFXXI exercise in February/March 1997, December 1996.

[12]304th Military Intelligence Battalion Alpha Company. Trojan SPIRIT II. Web page containing technical specifications of the Trojan SPIRIT II system. http://huachuca-link33.army.mil/, 1996.

[13]FORE Systems. ForeRunner ATM Backbone Switches ASX-200BX and ASX-1000. Web page containing technical specifications of the FORE ATM Switches. http://www.fore.com/products/swtch/index.html, 1996.

[14]et al Debra Hensgen. Original submission of smartnet scheduler paper to heterogeneous computing workshop. In Debra Hensgen, editor, *Proceedings Fifth Heterogeneous Computing Workshop.* The IEEE Computer Society Office of Naval Research, 1995.

[15]1996. Personal communication with Larry Levine, SRI.

[16]Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* The MIT Press, Cambridge, Massachusetts, 1990.

[17]Averill M. Law and W. David Kelton. *Simulation Modeling and Analysis, Second Edition.* McGraw-Hill, Inc., New York, 1991.

[18]John E. Hopcroft and Jeffrey D. Ullman. *Introduction To Automata Theory, Languages, and Computation.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1979.

[19]Allan Terry, Terry Barnes, and Rick Hayes-Roth. JTF ATD communications server architectural refinement through simulation experiments. Report concerning efforts to design a communications server which mediates all messaging to ensure most important communications are transmitted, September 1995.

[20]Teledesic Corp. ECHNICAL DETAILS OF THE TELEDESIC NETWORK . Web page containing technical specifications of the Teledesic System. http://www.teledesic.com/overview/system.html, 1997.

INITIAL DISTRIBUTION LIST

1.Defense Technical Information Center — 2
8725 John J. Kingman Road., Ste 0944
Ft. Belvoir, VA 22060-6218

2.Dudley Knox Library — 2
Naval Postgraduate School
411 Dyer Rd.
Monterey, CA 93943-5101

3.Director, Training and Education — 1
MCCDC, Code C46
1019 Elliot Road
Quantico, VA 22134-5027

4.Director, Marine Corps Research Center — 2
MCCDC, Code C40RC
2040 Broadway Street
Quantico, VA 22134-5107

5.Director, Studies and Analysis Division — 1
MCCDC, Code C45
3300 Russell Road
Quantico, VA 22134-5130

6.Marine Corps Representative — 1
Naval Postgraduate School
Code 037, Bldg. 234, HA-220
699 Dyer Road
Monterey, CA 93940

7.Marine Corps Tactical Systems Support Activity — 1
Technical Advisory Branch
Attn: Maj J.C. Cummiskey
Box 555171
Camp Pendelton, CA 92055-5080

8.Debra Hensgen — 5
Naval Postgraduate School
Code CS/Hd, Computer Sciences Dept.
833 Dyer Rd.
Monterey, CA 93943-5118

9.Geoffrey Xie 1
Naval Postgraduate School
Code CS/Xi, Computer Sciences Dept.
833 Dyer Rd.
Monterey, CA 93943-5118

10.Michael J. Lemanski 5
63 Spring Lake Drive
Stafford, VA 22554

11.Jesse C. Benton 2
14055 Old Kentucky Highway 1247
Waynesburg, KY 40489

12.ECJ6-NP 1
HQ USEUCOM
Unit 30400 Box 1000
APO AE 09128

13.Dr Roger Merk 1
NCCOSC RDTE Code D827
Bldg. 40, Rm 208A
53118 Gatchell Road
San Diego, CA 92152-7446

14.H.J. Siegel 1
Purdue University
Room 325, EE Building
School of Electrical and Computer Engineering
1285 Electrical Engineer Building
West Lafayette, IN 47907-1285

15.Richard Freund, Chief Scientist 1
Heterogeneous Computing Team
NCCOSC RDTE Div 4221 Rm 341A
53118 Gatchell Road
San Diego, CA 92152-7446

16.Dr. John J. Garstka 1
Scientific and Technical Advisor
The Joint Staff/J63-S&T
Pentagon Rm 1D777
Washington, DC 20318-6000

17.Dr. Robert J. Douglass 1
Program Manager
Information Integration, Information Systems Office
Defense Advance Research Projects
3701 North Fairfax Drive
Arlington, VA 22203-1714

18.Ray Glass 1
NCCOSC Code 7801
53118 Gatchell Road
San Diego, CA 92152-7446

19.Dr. Larry Levin 1
SRI International
Sarnoff Research Center
Princeton, NJ